

Introduction

INFORMATION IN THIS CHAPTER

- Book Overview and Key Learning Points
- Book Audience
- How This Book Is Organized
- Where to Go from Here

Pick your favorite cliché or metaphor you've heard regarding the Web. The aphorism might carry a generic description of Web security or generate a mental image of the threats and risks faced by and emanating from Web sites. This book attempts to cast a brighter light on the vagaries of Web security by tackling seven of the most, er, deadliest vulnerabilities that are exploited by attackers. Some of the attacks will sound very familiar. Other attacks may be unexpected, or seem uncommon simply because they aren't on a top 10 list or don't make headlines. Attackers often go for the lowest common denominator, which is why vulnerabilities such as cross-site scripting (XSS) and Structured Query Language (SQL) injection garner so much attention. Determined attackers also target the logic of a particular Web site – exploits that result in significant financial gain but have neither universal applicability from the attacker's perspective nor universal detection mechanisms for the defender.

On the Web, information equals money. Credit cards clearly have value to attackers; underground e-commerce sites have popped up that deal in stolen cards. Yet our personal information, passwords, e-mail accounts, online game accounts, all have value to the right buyer. Then consider economic espionage and state-sponsored network attacks. It should be possible to map just about any scam, cheat, trick, ruse, and other synonyms from real-world conflict between people, companies, and countries to an attack that can be accomplished on the Web. There's no lack of motivation for trying to gain illicit access to the wealth of information on the Web that isn't intended to be public.

BOOK OVERVIEW AND KEY LEARNING POINTS

Each chapter in this book presents examples of different attacks conducted against Web sites. The methodology behind the attack is explored, as well as showing its potential impact. Then the chapter moves on to address possible countermeasures

for different aspects of the attack. Countermeasures are a tricky beast. It's important to understand how an attack works before a good defense can be designed. It's also important to understand the limitations of a countermeasure and how other vulnerabilities might entirely bypass it. Security is an emergent property of the Web site; it's not a summation of individual protections. Some countermeasures will show up several times, and others make only a brief appearance.

BOOK AUDIENCE

Anyone who uses the Web to check e-mail, shop, or work will benefit from knowing how the personal information on those sites might be compromised or even how familiar sites can harbor malicious content. Although most security relies on the site's developers, consumers of Web applications can follow safe browsing practices to help protect their data.

Web application developers and security professionals will benefit from the technical details and methodology behind the Web attacks covered in this book. The first step to creating a more secure Web site is understanding the threats and risks of insecure code. Also, the chapters dive into countermeasures that can be applied to a site regardless of the programming language or technologies underpinning it.

Executive level management will benefit from understanding the threats to a Web site, and in many cases, how a simple attack – requiring nothing more than a Web browser – can severely impact a site. It should also illustrate that even though many attacks are simple to execute, good countermeasures require time and resources to implement properly. These points should provide strong arguments for allocating funding and resources to a site's security to protect the wealth of information that Web sites manage.

This book assumes some basic familiarity with the Web. Web security attacks manipulate HTTP traffic to inject payloads or take advantage of deficiencies in the protocol. They also require understanding HTML to manipulate forms or inject code that puts the browser at the mercy of the attacker. This isn't a prerequisite for understanding the broad strokes of an attack or learning how attackers compromise a site. For example, it's good to know that HTTP uses port 80 by default for unencrypted traffic and port 443 for traffic encrypted with the Secure Sockets Layer (SSL). Sites use the `https://` to designate SSL connections. Additional details are necessary for developers and security professionals who wish to venture deeper into the methodology of attacks and defense.

Readers already familiar with basic Web concepts can skip the next two sections.

One Origin to Rule Them All

Web browsers have gone through many iterations on many platforms: Konqueror, Mosaic, Mozilla, Internet Explorer, Opera, and Safari. Browsers have a rendering engine at their core. Microsoft calls IE's engine Trident. Safari uses WebKit. Firefox

relies on Gecko. Opera has Presto. These engines are responsible for rendering HTML into a Document Object Model, executing JavaScript, and ultimately providing the layout of a Web page.

The same origin policy (SOP) is a fundamental security border with the browser. The abilities and visibility of content is restricted to the origin that initially loaded the content. Unlike a low-budget horror movie where demons can come from one origin to wreak havoc on another, JavaScript is supposed to be restricted to the origin from whence it came. JavaScript's origin is the combination of at least the host name, port, and protocol of the containing page. In the age of mashups, this restriction is often considered an impediment to development. We'll revisit SOP several times, beginning with Chapter 1, Cross-Site Scripting.

Background Knowledge

This book is far too short to cover ancillary topics in detail. Several attacks and countermeasures dip into subjects such as cryptography with references to hashes, salts, symmetric encryption, and random numbers. Other sections venture into ideas about data structures, encoding, and algorithms. Sprinkled elsewhere are references to regular expressions. Effort has been made to introduce these concepts with enough clarity to show how they relate to a situation. Some suggested reading has been provided where more background knowledge is necessary or useful. Hopefully, this book will lead to more curiosity on such topics. A good security practitioner will be conversant on these topics even if mathematical or theoretical details remain obscure.

The most important security tool for this book is the Web browser. Quite often, it's the only tool necessary to attack a Web site. Web application exploits run the technical gamut of complex buffer overflows to single-character manipulations of the URI. The second most important tool in the Web security arsenal is a tool for sending raw HTTP requests. The following tools make excellent additions to the browser.

Netcat is the ancient ancestor of network security tools. It performs one basic function: open a network socket. The power of the command comes from the ability to send anything into the socket and capture the response. It is present by default on most Linux systems and MacOS X, often as the *nc* command. Its simplest use for Web security is as follows:

```
echo -e "GET / HTTP/1.0" | netcat -v mad.scientists.lab 80
```

Netcat has one failing for Web security tests: it doesn't support SSL. Conveniently, the *OpenSSL* command provides the same functionality with only minor changes to the command line. An example follows.

```
echo -e "GET / HTTP/1.0" | openssl s_client -quiet -connect mad.scientists.lab:443
```

Local proxies provide a more user-friendly approach to Web security assessment than command line tools because they enable the user to interact with the Web site as usual with a browser, but also provide a way to monitor and modify the traffic between a

browser and a Web site. The command line serves well for automation, but the proxy is most useful for picking apart a Web site and understanding what goes on behind the scenes of a Web request. The following proxies have their own quirks and useful features.

- Burp Proxy (www.portswigger.net/proxy/)
- Fiddler (www.fiddler2.com/fiddler2/), only for Internet Explorer
- Paros (<http://sourceforge.net/projects/paros/files/>)
- Tamper Data (<http://tamperdata.mozdev.org/>), only for Firefox

HOW THIS BOOK IS ORGANIZED

This book contains seven chapters that address a serious type of attack against Web sites and browsers alike. Each chapter provides an example of how an attack has been used against real sites before exploring the details of how attackers exploit the vulnerability. The chapters do not need to be tackled in order. Many attacks are related or build on one another in ways that make certain countermeasures ineffective. That's why it's important to understand different aspects of Web security, especially the concept that security doesn't end with the Web site, but extends to the browser as well.

Chapter 1: Cross-Site Scripting

Chapter 1 describes one of the most pervasive and easily exploited vulnerabilities that crop up in Web sites. XSS vulnerabilities are like the cockroaches of the Web, always lurking in unexpected corners of a site regardless of its size, popularity, or security team. This chapter shows how one of the most prolific vulnerabilities on the Web is exploited with nothing more than a browser and basic knowledge of HTML. It also shows how the tight coupling between the Web site and the Web browser can in fact be a fragile relationship in terms of security.

Chapter 2: Cross-Site Request Forgery

Chapter 2 continues the idea of vulnerabilities that target Web sites and Web browsers. CSRF attacks fool a victim's browser into making requests that the user didn't intend. These attacks are more subtle and difficult to block.

Chapter 3: Structured Query Language Injection

Chapter 3 turns the focus squarely onto the Web application and the database that drives it. SQL injection attacks are most commonly known as the source of credit-card theft. This chapter explains how many other exploits are possible with this simple vulnerability. It also shows that the countermeasures are relatively easy and simple to implement compared to the high impact successful attacks carry.

Chapter 4: Server Misconfiguration and Predictable Pages

Even the most securely coded Web site can be crippled by a poor configuration setting. This chapter explains how server administrators might make mistakes that expose the Web site to attack. This chapter also covers how the site's developers might also leave footholds for attackers by creating areas of the site where security is based more on assumption and obscurity than well-thought-out measures.

Chapter 5: Breaking Authentication Schemes

Chapter 5 covers one of the oldest attacks in computer security: brute force and the login prompt. Yet brute force attacks aren't the only way that a site's authentication scheme falls apart. This chapter covers alternate attack vectors and the countermeasures that will – and will not – protect the site.

Chapter 6: Logic Attacks

Chapter 6 covers a more interesting type of attack that blurs the line between technical prowess and basic curiosity. Attacks that target a site's business logic vary as much as Web sites do, but many have common techniques or target poor site designs in ways that can lead to direct financial gain for the attacker. This chapter talks about how the site is put together as a whole, how attackers try to find loopholes for their personal benefit, and what developers can do when faced with a problem that doesn't have an easy programming checklist.

Chapter 7: Web of Distrust

Chapter 7 brings Web security back to the browser. It covers the ways in which malicious software, malware, has been growing as a threat on the Web. This chapter also describes ways that users can protect themselves when the site's security is out of their hands.

WHERE TO GO FROM HERE

Hands-on practice provides some of the best methods for learning new security techniques or refining old ones. This book strives to provide examples and descriptions of the methodology for finding and preventing vulnerabilities. One of the best ways to reinforce this knowledge is by putting it to use against an actual Web application. It's unethical and usually illegal to start blindly flailing away at a random Web site of your choice. That doesn't limit the possibilities for practice. Scour sites such as SourceForge (www.sf.net/) for open-source Web applications. Download and install a few or a dozen. The act of deploying a Web site (and dealing with bugs in many of the applications) already builds experience with Web site concepts, programming patterns, and system administration that should be a foundation for

practicing security. Next, start looking for vulnerabilities in the application. Maybe it has an SQL injection problem or doesn't filter user-supplied input to prevent XSS. Don't always go for the latest release of a Web application; look for older versions that have bugs fixed in the latest version. You'll also have the chance to deal with different technologies, from PHP to Java to C#, from databases such as MySQL to Postgresql to Microsoft SQL Server. Also, you'll have access to the source code, so you can see why vulnerabilities arise, how a vulnerability may have been fixed between versions, or how you might fix the vulnerability. Hacking real applications (deployed in your own network) builds excellent experience.

Cross-Site Scripting

1

INFORMATION IN THIS CHAPTER

- Understanding HTML Injection
- Employing Countermeasures

When the Spider invited the Fly into his parlor, the Fly at first declined with the wariness of prey confronting its predator. The Internet is rife with traps, murky corners, and malicious hosts that make casually surfing random Web sites a dangerous proposition. Some areas are, if not obviously dangerous, at least highly suspicious. Web sites offering warez (pirated software), free porn, or pirated music tend to be laden with viruses and malicious software waiting for the next insecure browser to visit.

These Spiders' parlors also exist at sites typically assumed to be safe: social networking, well-established online shopping, Web-based e-mail, news, sports, entertainment, and more. Although such sites do not encourage visitors to download and execute untrusted virus-laden programs, they serve content to the browser. The browser blindly executes this content, a mix of Hypertext Markup Language (HTML) and JavaScript, to perform all sorts of activities. If you're lucky, the browser shows the next message in your inbox or displays the current balance of your bank account. If you're really lucky, the browser isn't siphoning your password to a server in some other country or executing money transfers in the background.

In October 2005, a user logged in to MySpace and checked out someone else's profile. The browser, executing JavaScript code it encountered on the page, automatically updated the user's own profile to declare someone named Samy their hero. Then a friend viewed that user's profile and agreed on his own profile that Samy was indeed "my hero." Then another friend, who had neither heard of nor met Samy, visited MySpace and added the same declaration. This pattern continued with such explosive growth that 24 hours later, Samy had over one million friends, and MySpace was melting down from the traffic. Samy had crafted a cross-site scripting (XSS) attack that, with approximately 4,000 characters of text, caused a denial

of service against a company whose servers numbered in the thousands and whose valuation at the time flirted around \$500 million. The attack also enshrined Samy as the reference point for the mass effect of XSS. (An interview with the creator of Samy can be found at <http://blogoscoped.com/archive/2005-10-14-n81.html>.)

How often have you encountered a prompt to reauthenticate to a Web site? Have you used Web-based e-mail? Checked your bank account online? Sent a tweet? Friendied someone? There are examples of XSS vulnerabilities for every one of these Web sites.

XSS isn't always so benign that it acts merely as a nuisance for the user. (Taking down a Web site is more than a nuisance for the site's operators.) It is also used to download keyloggers that capture banking and online gaming credentials. It is used to capture browser cookies to access victims' accounts with the need for a username or password. In many ways, it serves as the stepping stone for very simple, yet very dangerous attacks against anyone who uses a Web browser.

UNDERSTANDING HTML INJECTION

XSS can be more generally, although less excitingly, described as HTML injection. The more popular name belies the fact that successful attacks need not cross sites or domains and need not consist of JavaScript to be effective.

An XSS attack rewrites the structure of a Web page or executes arbitrary JavaScript within the victim's Web browser. This occurs when a Web site takes some piece of information from the user – an e-mail address, a user ID, a comment to a blog post, a zip code, and so on – and displays the information in a Web page. If the Web site is not careful, then the meaning of the HTML document can be disrupted by a carefully crafted string.

For example, consider the search function of an online store. Visitors to the site are expected to search for their favorite book, movie, or pastel-colored squid pillow, and if the item exists, they purchase it. If the visitor searches for DVD titles that contain *living dead*, the phrase might show up in several places in the HTML source. Here, it appears in a meta tag.

```
<SCRIPT LANGUAGE="JavaScript" SRC="/script/script.js"></SCRIPT>
<meta name="description" content="Cheap DVDs. Search results for
  living dead" />
<meta name="keywords" content="dvds,cheap,prices" /><title>
```

However, later the phrase may be displayed for the visitor at the top of the search results, and then near the bottom of the HTML inside a script element that creates an ad banner.

```
<div>matches for "<span id="ct100_body_ct100_lblSearchString">
  living dead</span>"</div>
...lots of HTML here...
<script type="text/javascript"><!--
  ggl_ad_client = "pub-6655321";
```

```

ggl_ad_width = 468;
ggl_ad_height = 60;
ggl_ad_format = "468x60_as";

ggl_ad_channel = "";
ggl_hints = "living dead";
//-->
</script>

```

XSS comes in to play when the visitor can use characters normally reserved for HTML markup as part of the search query. Imagine if the visitor appends a double quote (“) to the phrase. Compare how the browser renders the results of the two different queries in each of the windows in Figure 1.1.

Note that the first result matched several titles in the site’s database, but the second search reported “No matches found” and displayed some guesses for a close match. This happened because *living dead*” (with quote) was included in the database query and no titles existed that ended with a quote. Examining the HTML source of the response confirms that the quote was preserved:

```

<div>matches for "<span id="ct100_body_ct100_1b1SearchString">
  living dead"</span>"</div>

```

If the Web site will echo anything we type in the search box, what might happen if a more complicated phrase were used? Figure 1.2 shows what happens when JavaScript is entered directly into the search.

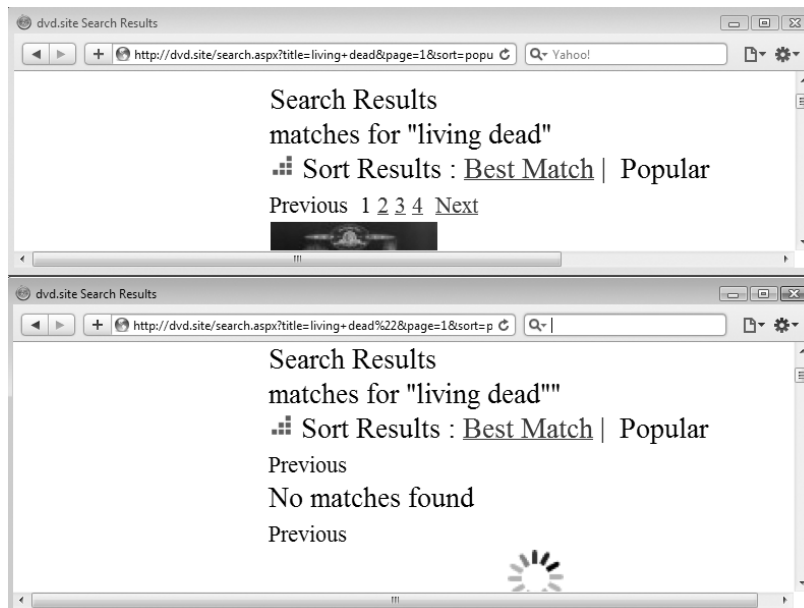


FIGURE 1.1

Search Results with and without a Tailing Quote (“)

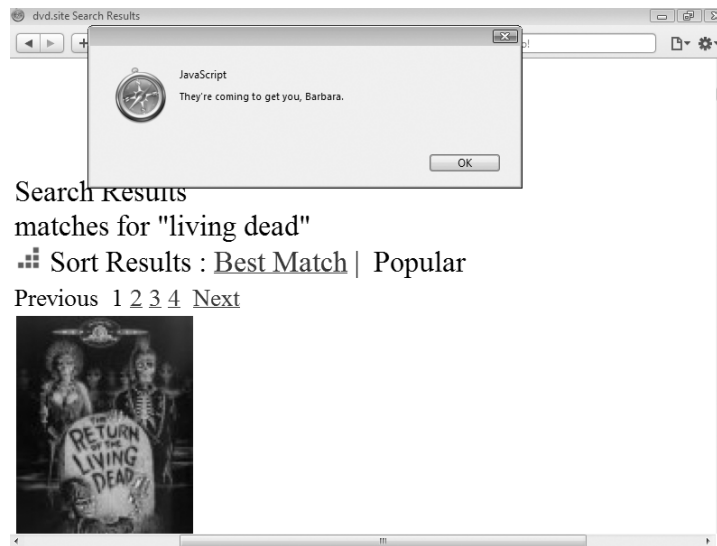


FIGURE 1.2

An Ominous Warning Delivered via XSS

By breaking down the search phrase, we see how the page was rewritten to convey a very different message to the Web browser than the Web site's developers intended. The HTML language is a set of grammar and syntax rules that inform the browser how to interpret pieces of the page. The rendered page is referred to as the Document Object Model (DOM). The use of quotes and angle brackets enabled the attacker to change the page's grammar to add a JavaScript element with code that launched a pop-up window. This happened because the phrase was placed directly in line with the rest of the HTML content.

```
<div>matches for "<span id="ct100_body_ct100_lblSearchString">
  living dead<script>alert("They're coming to get you, Barbara.")
</script></span>"</div>
```

Instead of displaying `<script>alert...` as text like it does for the words *living dead*, the browser sees the `<script>` tag as the beginning of a code block and renders it as such. Consequently, the attacker is able to arbitrarily change the content of the Web page by manipulating the DOM.

Before we delve too deeply into what an attack might look like, let's see what happens to the phrase when it appears in the meta tag and ad banner. Here is the meta tag when the phrase *living dead* is used:

```
<meta name="description" content="Cheap DVDs. Search results for
  living dead&quot;" />
```

The quote character has been rewritten to its HTML-encoded version – `"` – which browsers know to display as the “ symbol. This encoding preserves the syntax

of the meta tag and the DOM in general. Otherwise, the syntax of the meta tag would have been slightly different:

```
<meta name="description" content="Cheap DVDs. Search results for
  living dead" />
```

This lands an innocuous pair of quotes inside the element and most browsers will be able to recover from the apparent typo. On the other hand, if the search phrase is echoed verbatim in the meta element's *content* attribute, then the attacker has a delivery point for an XSS payload:

```
<meta name="description" content="Cheap DVDs. Search results for
  living dead"/>
<script>alert("They're coming to get you, Barbara.")</script>
<meta name="" />
```

Here's a more clearly annotated version of the XSS payload. Note how the syntax and grammar of the HTML page have been changed. The first meta element is properly closed, a script element follows, and a second meta element is added to maintain the validity of the HTML.

```
<meta name="description" content="Cheap DVDs. Search results for
  living dead"/>   close content attribute with a quote, close
  the meta element with />
<script>...</script>   add some arbitrary JavaScript
<meta name=""   create an empty meta element to prevent the browser
  from displaying the dangling "/> from the original <meta
  description... element
" />
```

The `ggl_hints` parameter in the ad banner script element can be similarly manipulated. Yet, in this case, the payload already appears inside a script element, so the attacker needs only to insert valid JavaScript code to exploit the Web site. No new elements needed to be added to the DOM for this attack. Even if the developers had been savvy enough to blacklist `<script>` tags or any element with angle brackets, the attack would have still succeeded.

```
<script type="text/javascript"><!--
  ggl_ad_client = "pub-6655321";
  ggl_ad_width = 468;
  ggl_ad_height = 60;
  ggl_ad_format = "468x60_as";

  ggl_ad_channel = "";
  ggl_hints = "living dead";   close the ggl_hints string with";
  ggl_ad_client="pub-attacker";   override the ad_client to give
  the attacker credit
  function nefarious() { }   perhaps add some other function
  foo="   create a dummy variable to catch the final ";
  ";
  //-->
</script>
```

Each of the previous examples demonstrated an important aspect of XSS attacks: the location on the page where the payload is echoed influences what characters are necessary to implement the attack. In some cases, new elements can be created, such as `<script>` or `<iframe>`. In other cases, an element's attribute might be modified. If the payload shows up within a JavaScript variable, then the payload need only consist of code.

Pop-up windows are a trite example of XSS. More vicious payloads have been demonstrated to

- Steal cookies so attackers can impersonate victims without having to steal passwords
- Spoof login prompts to steal passwords (attackers like to cover all the angles)
- Capture keystrokes for banking, e-mail, and game Web sites
- Use the browser to port scan a local area network
- Surreptitiously reconfigure a home router to drop its firewall
- Automatically add random people to your social network
- Lay the groundwork for a cross-site request forgery (CSRF) attack

Regardless of what the actual payload is trying to accomplish, all forms of the XSS attack rely on the ability of a user-supplied bit of information to be rendered in the site's Web page such that the DOM structure will be modified. Keep in mind that changing the HTML means that the Web site is merely the penultimate victim of the attack. The Web site acts as a broker that carries the payload from the attacker to the Web browser of anyone who visits it.

Alas, this chapter is far too brief to provide a detailed investigation of all XSS attack techniques. One in particular deserves mention among the focus on inserting JavaScript code and creating HTML elements, but is addressed here only briefly: Cascading Style Sheets (CSS). Cascading Style Sheets, abbreviated CSS and not to be confused with this attack's abbreviation, control the layout of a Web site for various media. A Web page could be resized or modified depending on whether it's being rendered in a browser, a mobile phone, or sent to a printer. Clever use of CSS can attain much of the same outcomes as a JavaScript-based attack. In 2006, MySpace suffered a CSS-based attack that tricked victims into divulging their passwords (www.caughq.org/advisories/CAU-2006-0001.txt). Other detailed examples can be found at <http://p42.us/css/>.

Identifying Points of Injection

The Web browser is not to be trusted. Obvious sources of attack may be links or form fields. Yet, all data from the Web browser should be considered tainted. Just because a value is not evident, such as the User-Agent header that identifies every type of browser, it does not mean that the value cannot be modified by a malicious user. If the Web application uses some piece of information from the browser, then that information is a potential injection point regardless of whether the value is assumed to be supplied manually by a human or automatically by the browser.

Uniform Resource Identifier Components

Any portion of the Uniform Resource Identifier (URI) can be manipulated for XSS. Directory names, file names, and parameter name/value pairs will all be interpreted by the Web server in some manner. The URI parameters may be the most obvious area of concern. We've already seen what may happen if the search parameter contains an XSS payload. The URI is dangerous even when it might be invalid, point to a nonexistent page, or have no bearing on the Web site's logic. If the Web site echos the link in a page, then it has the potential to be exploited. For example, a site might display the URI if it can't find the location the link was pointing to.

```
<html>
Oops! We couldn't find 
```

Another common Web design pattern is to place the previous link in an anchor element, which has the same potential for mischief.

```
<a href=" http://some.site/home/index.php? ="><script></script>
<foo a="">search again</a>
```

Form Fields

Forms collect information from users, which immediately make the supplied data potentially tainted. This obviously applies to the fields users are expected to fill out, such as login name, e-mail address, or credit-card number. Less obvious are the fields that users are not expected to modify, such as input type=*hidden* or input fields with the *disable* attribute. Any form field's value can be trivially modified before it is submitted to the server. Considering client-side security as secure is a mistake that naive or unaware developers will continue to make.

Hypertext Transfer Protocol Request Headers

Every browser includes certain Hypertext Transfer Protocol (HTTP) headers with each request. Everything from the browser can be spoofed or modified. Two of the most common headers used for successful injections are the User-Agent and Referer. If the Web site parses and displays any HTTP client headers, then it should sanitize them.

User-Generated Content

Binary contents such as images, movies, or PDF files may carry embedded JavaScript or other code that could be executed within the browser. Content-sharing sites thrive on users uploading new items. Attacks delivered via these mechanisms may be less common, but they are no less of a threat. See the Section, "Subverting Multipurpose Internet Mail Extensions Types," discussed later in this chapter for more details about how such files can be subverted.

JavaScript Object Notation

JavaScript Object Notation (JSON) is a method for representing arbitrary JavaScript data types as a string safe for HTTP communications. A Web-based e-mail site might use JSON to retrieve e-mail messages or contact information. In 2006, Gmail

had a very interesting CSRF, an attack to be explained in Chapter 2, “Cross-Site Request Forgery,” identified in its JSON-based contact list handling (<http://googlified.com/follow-up-on-the-gmail-bug/>). An e-commerce site might use JSON to track product information. Data may come into JSON from one of the previously mentioned vectors (URI parameters, form fields, etc.). The peculiarities of passing content through JSON parsers and `eval()` functions bring a different set of security concerns because of the ease with which JavaScript objections and functions can be modified. The best approach to protecting sites that use JSON is to rely on JavaScript development frameworks. These frameworks not only offer secure methods for handling untrusted content but they also have extensive unit tests and security-conscious developers working on them. Well-tested code alone should be a compelling reason for adopting a framework rather than writing one from scratch. Table 1.1 lists several popular frameworks that will aid the development of sites that rely on JSON and the `xmlHttpRequestObject` for data communications between the browser and the Web site.

These frameworks focus on creating dynamic, highly interactive Web sites. They do not secure the JavaScript environment from other malicious scripting content. See the Section, “JavaScript Sandboxes,” for more information on securing JavaScript-heavy Web sites.

DOM Properties

An interesting XSS delivery variant uses the DOM to modify itself in an unexpected manner. The attacker assigns the payload to some property of the DOM that will be read and echoed by a script within the same Web page. A nice example is Bugzilla bug 272620. When a Bugzilla page encountered an error, its client-side JavaScript would create a user-friendly message:

```
document.write("<p>URL: " + document.location + "</p>")
```

If the `document.location` property of the DOM could be forced to contain malicious HTML, then the attacker would succeed in exploiting the browser. The `document.location` property contains the URI used to request the page and hence it is easily modified by the attacker. The important nuance here is that the server need not know or write the value of `document.location` into the Web page. The attack occurs

Table 1.1 Common JavaScript development frameworks

Framework	Project home page
Dojo	www.dojotoolkit.org/
Direct Web Remoting	http://directwebremoting.org/
Google Web Toolkit	http://code.google.com/webtoolkit/
MooTools	http://mootools.net/
jQuery	http://jquery.com/
Prototype	www.prototypejs.org/
YUI	http://developer.yahoo.com/yui/

purely in the Web browser when the attacker crafts a malicious URI, perhaps adding script tags as part of the query string like so:

- `http://bugzilla/enter_bug.cgi?<script>...</script>`

The malicious URI causes Bugzilla to encounter an error that causes the browser, via the `document.write` function, to update its DOM with a new paragraph and script elements. Unlike the other forms of XSS delivery, the server did not echo the payload to the Web page. The client unwittingly writes the payload from the `document.location` into the page.

```
<p>URL: http://bugzilla/enter_bug.cgi?<script>...</script></p>
```

NOTE

The countermeasures for XSS injection, via DOM properties, require client-side validation. Normally, client-side validation is not emphasized as a countermeasure for any Web attack. This is exceptional because the attack occurs purely within the browser and cannot be influenced by any server-side defenses. Modern JavaScript development frameworks, when used correctly, offer relatively safe methods for querying properties and updating the DOM. At the very least, frameworks provide a centralized code library that is easy to update when vulnerabilities are identified.

Distinguishing Different Delivery Vectors

Because XSS uses a compromised Web site as a delivery mechanism to a browser, it is necessary to understand not only how a payload enters the Web site but also how and where the site renders the payload for the victim's browser. Without a clear understanding of where potentially malicious user-supplied data may appear, a Web site may have inadequate security or an inadequate understanding of the impact of a successful exploit.

Reflected

Reflected XSS is injected and observed in a single HTTP request/response pair. For example, pages in a site that provide search typically redisplayed “you searched for foobar.” Instead of searching for foobar, you search for `<script>destroyAllHumans()</script>` and watch as the JavaScript is reflected in the HTTP response. Reflected XSS is stateless. Each search query returns a new page with whatever attack payload or search term was used. The vulnerability is a one-to-one reflection. The browser that submitted the payload will be the browser that is affected by the payload. Consequently, attack scenarios typically require the victim to click on a precreated link. This might require some simple social engineering along the lines of “check out the pictures I found on this link” or be as simple as hiding the attack behind a URI shortener. The search examples in the previous section demonstrated reflected XSS attacks.

Persistent

Persistent XSS vulnerabilities have the benefit (from the attacker's perspective) for enabling a one-to-many attack. The attacker need deliver a payload once, and then wait for victims to visit the page where the payload manifests. Imagine a shared

calendar in which the title of a meeting includes the XSS payload. Anyone who views the calendar would be affected by the XSS payload. Both reflected and persistent XSS are dangerous. A persistent payload might also be injected on one page of the Web site and displayed on another. For example, reflected XSS might show up in the search function of a Web site. A persistent XSS could appear if the site also had a different page that tracked the most recent or most popular searches for other users to view.

Higher Order

Higher order XSS occurs when a payload is injected in one application, but manifests in a separate Web site. Imagine a Web site, Alpha, that collects and stores the User-Agent string of every browser that visits it. This string is stored in a database but is never used by the Alpha site. Site Bravo, on the other hand, takes this information and displays the unique User-Agent strings. Site Bravo, pulling values from the database, might assume that input validation isn't necessary because the database is a trusted source. (The database is a trusted source because it will not manipulate or modify data, but it contains data that have already been tainted.)

For a better example of higher order XSS, try searching for “<title><script>” in any search engine. Search engines commonly use the <title> element to label Web pages in their search results. If the engine indexed a site with a malicious title and failed to encode its content properly, then an unsuspecting user could be compromised by doing nothing more than querying the search engine. The search in Figure 1.3

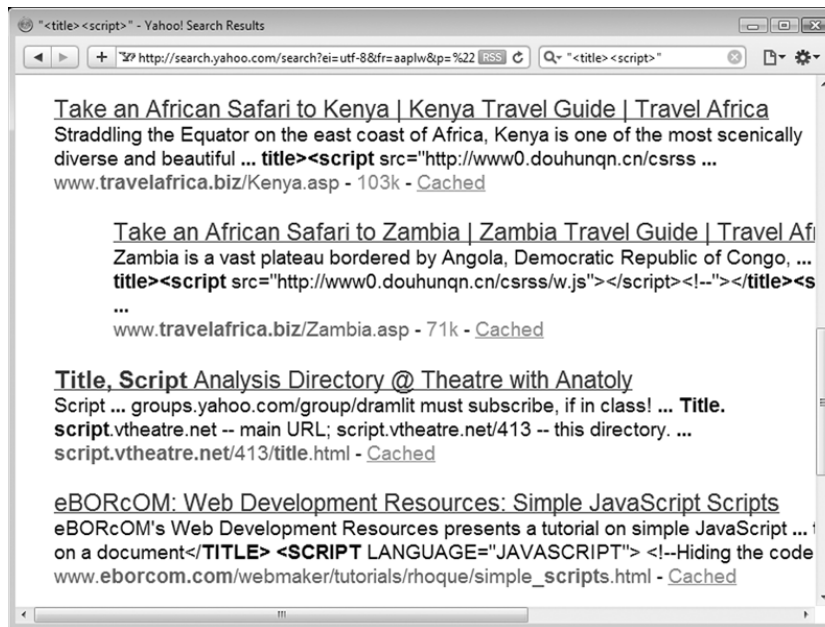


FIGURE 1.3

Plan a Trip to Africa – While Your Browser Visits China

was safe, mainly because the title tags were encoded to prevent the script tags from executing.

Handling Character Sets Safely

Although English is currently the most pervasive language throughout Web sites on the Internet, other languages such as Chinese (Mandarin), Spanish, Japanese, and French hold a significant share. (I would cite a specific reference for this list of languages, but the Internet being what it is, the list could easily be surpassed by lolcat, l33t, or Klingon by the time you read this – none of which invalidates the problem of character encoding.) Consequently, Web browsers must be able to support non-English writing systems whether the system merely includes accented characters, ligatures, or complex ideograms. One of the most common encoding schemes used on the Web is the UTF-8 standard.

Character encoding is a complicated, often convoluted, process that Web browsers have endeavored to support as fully as possible. Combine any complicated process that evolves over time with software that aims for backward compatibility, and you arrive at quirks like UTF-7 – a widely supported, nonstandard encoding scheme.

This meandering backstory finally brings us to using character sets for XSS attacks. Most payloads attempt to create an HTML element such as `<script>` in the DOM. A common defensive programming measure strips the potentially malicious angle brackets (`<` and `>`) from any user-supplied data, and thus crippling `<script>` and `<iframe>` elements to become innocuous text. UTF-7 provides an alternate encoding for the angle brackets: `+ADw-` and `+AD4-`.

The `+` and `-` indicate the start and stop of the encoded sequence (also called *Unicode-shifted encoding*). So, any browser that can be instructed to decode the text as UTF-7 will turn the `+ADw-script+AD4-` characters into `<script>` when rendering the HTML.

The key is to force the browser to accept the content as UTF-7. Browsers rely on Content-Type HTTP headers and HTML meta elements for instructions on which character set to use. When an explicit content-type is missing, the browser's decision on how to interpret the characters is vague.

This HTML example shows how a page's character set is modified by a meta tag. Figure 1.4 shows how a browser renders the page, including the uncommon syntax for the script tags.

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-7">
</head>
<body>
+ADw-script+AD4-alert("Just what do you think you're doing,
    Dave?")+ADw-/script+AD4-
</body>
</html>
```

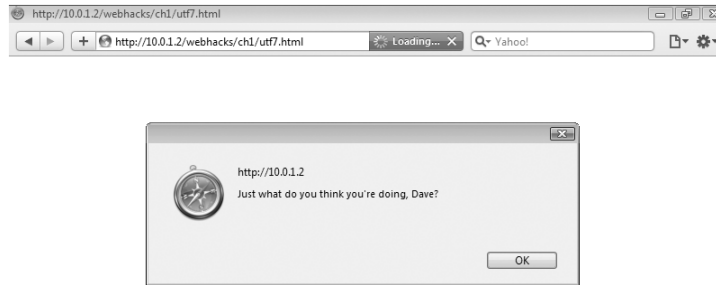


FIGURE 1.4
Creating `<script>` Tags with Alternate Character Sets

UTF-7 demonstrates a specific type of attack, but the underlying problem is due to the manner in which Web application handles characters. This UTF-7 attack can be fixed by forcing the encoding scheme of the HTML page to be UTF-8 (or some other explicit character set) in the HTTP header:

```
Date: Sun, 13 Sep 2009 00:47:44 GMT
Content-Type: text/html;charset=utf-8
Connection: keep-alive
Server: Apache/2.2.9 (Unix)
```

Or with a meta element:

```
<meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
```

This just addresses one aspect of the vulnerability. Establishing a single character set doesn't absolve the Web site of all vulnerabilities, and many XSS attacks continue to take advantage of poorly coded sites. The encoding scheme itself isn't the problem. The manner in which the site's programming language and software libraries handle characters are where the true problem lies, as demonstrated in the next sections.

Attack Camouflage with Percent Encoding

First some background. Web servers and browsers communicate by shuffling characters (bytes) back and forth between them. Most of the time, these bytes are just letters, numbers, and punctuation that make up HTML, e-mail addresses, blog posts about cats, flame wars about the best Star Wars movie, and so on. An 8-bit character produces 255 possible byte sequences. HTTP only permits a subset of these to be part of a request but provides a simple solution to write any character if necessary: percent encoding. Percent encoding (also known as URI or URL encoding) is simple. Take the ASCII value in hexadecimal of the character, prepend the percent sign (%), and send. For example, the lowercase letter z's hexadecimal value is 0x7a and would be encoded in a URI as %7a. The word "zombie" becomes %7a%6f%6d%62%69%65. RFC 3986 describes the standard for percent encoding.

Percent encoding attacks aren't relegated to characters that must be encoded in an HTTP request. Encoding a character with special meaning in the URI can lead to profitable exploits. Two such characters are the dot (.) and forward slash (/). The dot is used to delineate a file suffix, which might be handled by the Web server in a specific manner, for example, .php is handled by a PHP engine, .asp by IIS, and .py by a Python interpreter.

A simple example dates back to 1997, when the l0pht crew published an advisory for IIS 3.0 (www.securityfocus.com/bid/1814/info). The example might bear the dust of over a decade (after all, Windows 2000 didn't yet exist and Mac OS was pre-Roman numeral with version 8), but the technique remains relevant today. The advisory described an absurdly simple attack: replace the dot in a file suffix with the percent encoding equivalent, %2e, and IIS would serve the source of the file rather than its interpreted version. Consequently, requesting /login%2easp instead of /login.asp would reveal the source code of the login page. That's a significant payoff for a simple hack.

In other words, the Web server treated login%2easp differently from login.asp. This highlights how a simple change in character can affect the code path in a Web application. In this case, it seemed that the server decided how to handle the page before decoding its characters. We'll see more examples of this Time of Check, Time of Use (TOCTOU) problem. It comes in quite useful for bypassing insufficient XSS filters.

Encoding 0x00 – Nothing Really Matters

Character set attacks against Web applications continued to proliferate in the late 1990s. The NULL-byte attack was described in the *Perl CGI problems* article in Phrack issue 55 (www.phrack.org/issues.html?issue=55&id=7#article). Most programming languages use NULL to represent “nothing” or “empty value” and treat a byte value of 0 (zero) as NULL. The basic concept of this attack is to use a NULL character to trick a Web application into processing a string differently than the programmer intended.

The earlier example of percent encoding the walking dead (%7a%6f%6d%62%69%65) isn't particularly dangerous, but dealing with control characters and the NULL byte can be. The NULL byte is simply 0 (zero) and is encoded as %00. In the C programming language, which underlies most operating systems and programming languages, the NULL byte terminates a character string. So a word like zombie is internally represented as 7a6f6d62696500. For a variety of reasons, not all programming languages store strings in this manner.

You can print strings in Perl by using hex values:

```
$ perl -e 'print "\x7a\x6f\x6d\x62\x69\x65"'
```

Or in Python:

```
$ python -c 'print "\x7a\x6f\x6d\x62\x69\x65"'
```

Each happily accepts NULL values in a string:

```
$ perl -e 'print "\x7a\x6f\x6d\x62\x69\x65\x00\x41"'
zombieA
$ python -c 'print "\x7a\x6f\x6d\x62\x69\x65\x00\x41"'
zombieA
```

To prove that each considers NULL as part of the string rather than a terminator, here is the length of the string and an alternate view of the output:

```
$ perl -e 'print length("\x7a\x6f\x6d\x62\x69\x65\x00\x41")'
8
$ perl -e 'print "\x7a\x6f\x6d\x62\x69\x65\x00\x41"' | cat -tve
zombie^@A$
$ python -c 'print len("\x7a\x6f\x6d\x62\x69\x65\x00\x41")'
8
$ python -c 'print "\x7a\x6f\x6d\x62\x69\x65\x00\x41"' | cat -tve
zombie^@A$
```

A successful attack relies on the Web language to carry around this NULL byte until it performs a task that relies on a NULL-terminated string, such as opening a file. This can be easily demonstrated on the command line with Perl. On a Unix or Linux system, the following command will be used, in fact, to open the `/etc/passwd` file instead of the `/etc/passwd.html` file.

```
$ perl -e '$s = "/etc/passwd\x00.html"; print $s; open(FH,"<$s");
while(<FH>) { print }'
```

The reason that `%00` (NULL) can be an effective attack is that Web developers may have implemented security checks that they believe will protect the Web site even though the check can be trivially bypassed. The following examples show what might happen if the attacker tries to access the `/etc/passwd` file. The URI might load a file referenced in the `s` parameter as in

- `http://site/page.cgi?s=/etc/passwd`

The Web developer could block any file that doesn't end with `".html"` as shown in this simple command:

```
$ perl -e '$s = "/etc/passwd"; if ($s =~ m/\.html$/) { print
"match" } else { print "block" }'
block
```

On the other hand, the attacker could tack `"%00.html"` on to the end of `/etc/passwd` to bypass the file suffix check.

```
$ perl -e '$s = "/etc/passwd\x00.html"; if ($s =~ m/\.html$/)
{ print "match" } else { print "block" }'
match
```

Instead of looking for a file suffix, the Web developer could choose to always append one. Even in this case, the attempted security will fail because the attacker

can submit still “/etc/passwd%00” as the attack and the string once again become “/etc/passwd%00.html,” which we’ve already seen gets truncated to /etc/passwd when passed into the open() function.

Alternate Encodings for the Same Character

Character encoding problems stretch well beyond unexpected character sets, such as UTF-7, and NULL characters. We’ll leave the late 1990s and enter 2001 when the “double decode” vulnerability was reported for IIS (MS01-026, www.microsoft.com/technet/security/bulletin/MS01-026.mspx). Exploits against double decode targeted the UTF-8 character set and focused on very common URI characters. The exploit simply rewrote the forward slash (/) with a UTF-8 equivalent using an over-long sequence, %c0%af.

This sequence could be used to trick IIS into serving files that normally would have been restricted by its security settings, whereas <http://site/../../../../windows/system32/cmd.exe> would normally be blocked, rewriting the slashes in the directory traversal would bypass security:

- <http://site/../../../../windows/system32/cmd.exe>

Once again the character set has been abused to compromise the Web server. Even though this particular issue was analyzed in detail, it resurfaced in 2009 in Microsoft’s advisory 971492 (www.microsoft.com/technet/security/advisory/971492.mspx). A raw HTTP request for this vulnerability would look like:

```
GET /../../../../protected/protected.zip HTTP/1.1 Translate:
f Connection: close Host:
```

Why Encoding Matters for XSS

The previous discussions of percent encoding detoured from XSS with demonstrations of attacks against the Web application’s programming language (for example, Perl, Python, and %00) or against the server itself (IIS and %c0%af). We’ve taken these detours along the characters in a URI to emphasize the significance of using character encoding schemes to bypass security checks. Instead of special characters in the URI (dot and forward slash), consider some special characters used in XSS attacks:

```
<script>maliciousFunction(document.cookie)</script>
onLoad=maliciousFunction()
javascript:maliciousFunction()
```

The angle brackets (< and >), quotes, and parentheses are the usual prerequisites for an XSS payload. If the attacker needs to use one of those characters, then the focus of the attack will switch to using control characters such as NULL and alternate encodings to bypass the Web site’s security filters.

Probably the most common reason XSS filters fail is that the input string isn’t correctly normalized.

Not Failing Secure

Even carefully thought out, protections can be crippled by unexpected behavior in the application's framework.

The earlier examples using overlong encoding (a sequence that starts with `%c0`) showed how UTF-8 could create alternate sequences for the same character. There are a handful of other bytes that if combined with an XSS payload can wreak havoc on a Web site. For example, UTF-8 sequences are not supposed to start with `%fe` or `%ff`. The UTF-8 standard describes situations where the `%fe%ff` sequence should be forbidden, as well as situations when it may be allowed. The special sequence `%ff%fd` indicates a replacement character – used when an interpreter encounters an unexpected or illegal sequence. In fact, current UTF-8 sequences are supposed to be limited to a maximum of bytes to represent a character, which would forbid sequences starting with `%f5` or greater.

So, what happens when the character set interpreter meets one of these bytes? It depends. A function may silently fail on the character and continue to interpret the string, perhaps comparing it with a whitelist. Or the function may stop at the character and not test the remainder of the string for malicious characters.

WARNING

Payloads may also be disguised with invalid character sequences. The two byte sequence `%80%22` might cause a parser to believe it represents a single multiple-width character, but a browser might consider the bytes as two individual characters, which means that `%22` – a quote character – would have been sneaked through a filter.

Avoiding Blacklisted Characters Altogether

XSS exploits typically rely on JavaScript to be most effective. Simple attacks require several JavaScript syntax characters to work. Payloads that use strings require quotes – at least the pedestrian version `alert('foo')` does. Single quotes also show up in SQL injection payloads. This notoriety has put the single quote on many a Web site's list of forbidden input characters. The initial steps through the input validation minefield try encoded variations of the quote character. Yet, these don't always work.

HTML elements don't require spaces to delimit their attributes.

```
<img/src="."alt=""onerror="alert('zombie')"/>
```

JavaScript doesn't have to rely on quotes to establish strings, nor do HTML attributes like `src` and `href` require them.

```
alert(String.fromCharCode(62,72,61,69,6e,73,21));
alert(/flee puny humans/.source);
alert(((function(){/*sneaky little hobbitses*/}).toString().
  substring(15,38));
<iframe src=//site/page>
```

The JavaScript language continues to evolve. None of the previous techniques exploits a deficiency of the language; they're all valid constructions (if the browser executes it, then it must be valid!). As new objects and functions extend the language, it's safe to assume that some of them will aid XSS payload obfuscation and shortening. Keeping an exclusion list up-to-date is a daunting task for the current state-of-the-art XSS. Knowing that more techniques will come only highlights the danger of placing too much faith in signatures to identify and block payloads.

Dealing with Browser Quirks

Web browsers face several challenges when dealing with HTML. Most sites attempt to adhere to the HTML4 standard, but some browsers extend standards for their own purposes or implement them in subtly different ways. Added to this mix are Web pages written with varying degrees of correctness, typos, and expectations of a particular browser's quirks.

The infamous SAMY MySpace XSS worm relied on a quirky behavior of Internet Explorer's handling of spaces and line feeds within a Web page. Specifically, part of the attack broke the word "javascript" into two lines:

```
style="background:url('java  
script:eval(...
```

Browser quirks are an insidious problem for XSS defenses. A rigorous input filter might be tested and considered safe, only to fail when confronted with a particular browser's implementation. For example, an attacker may target a particular browser by creating payloads with

- Invalid sequences, `java%feffscript`
- Alternate separator characters, `href=#%18%0eonclick=maliciousFunction()`
- Whitespace characters like tabs (`0x09` or `0x0b`) and line feed (`0x0a`) in a reserved word, `java[0x0b]script`
- Browser-specific extensions, `-moz-binding: url(...)`

This highlights how attackers can elude pattern-based filters (for example, reject "javascript" anywhere in the input). For developers and security testers, it highlights the necessity to test countermeasures in different browser versions to avoid problems due to browser quirks.

The Unusual Suspects

The risk of XSS infection doesn't end once the Web site has secured itself from malicious input, modified cookies, and character encoding schemes. At its core, an XSS attack requires the Web browser to interpret some string of text as JavaScript. To this end, clever attackers have co-opted binary files that would otherwise seem innocuous.

In March 2002, an advisory was released for Netscape Navigator that described how image files, specifically the GIF or JPEG formats, could be used to deliver malicious

JavaScript (<http://security.FreeBSD.org/advisories/FreeBSD-SA-02:16.netscape.asc>). These image formats include a text field for users (and programs and devices) to annotate the image. For example, tools such as Photoshop and the Gnu Image Manipulation Program (GIMP) insert default strings. Modern cameras will tag the picture with the date and time it was taken – even the camera’s current GPS coordinates if so enabled.

The researcher discovered that Navigator can actually treat the text within the image’s comment field as potential HTML. Consequently, an image with the comment `<script>alert('Open the pod bay doors please, Hal.')` would cause the browser to launch the pop-up window.

Once again, let yourself imagine that an eight-year-old vulnerability is no longer relevant, and consider this list of XSS advisories in files that might otherwise be considered safe.

- XSS vulnerability in Macromedia Flash ad user tracking capability allows remote attackers to insert arbitrary Javascript via the clickTAG field, April 2003 (<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0208>).
- Universal XSS in PDF files, December 2006 (http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf).
- XSS in Safari RSS reader, January 2009 (<http://brian.mastenbrook.net/display/27>).
- Adobe Flex 3.3 SDK DOM-Based XSS, August 2009. Strictly speaking, this is still an issue with generic HTML. The point to be made concerns relying on an SDK to provide a secure code (<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1879>).

Subverting MIME Types

Web browsers are written with the best intentions of providing correct content to users even if some extra whitespace might be present in an HTML tag or the reported MIME type of a file doesn’t line up with its actual type. Early versions of the Internet Explorer examined the first 200 bytes of a file to help determine how it should be presented. Common file types have magic numbers – preambles or predefined bytes that indicate their type and even version. So, even if a PNG file starts off with a correct magic number (hexadecimal 89504E470D0A1A0A) but contains HTML markup within the first 200 bytes, then Internet Explorer (IE) might consider the image to be HTML and execute it accordingly.

This problem is not specific to Internet Explorer. All Web browsers use some variation of this method to determine how to render an unknown, vague, or unexpected file type.

MIME-type subversion isn’t a common type of attack because it can be mitigated by diligent server administrators who configure the Web server to explicitly – and correctly – describe a file’s MIME type. Nevertheless, it represents yet another situation where the security of the Web site is at the mercy of a browser’s quirks. MIME-type detection is described in RFC 2936, but there is not a common standard identically implemented by all browsers. Keep an eye on HTML5 section 4.2

(<http://dev.w3.org/html5/spec/Overview.html>) and the draft specification (<http://tools.ietf.org/html/draft-abarth-mime-sniff-01>) for progress in the standardization of this feature.

EMPLOYING COUNTERMEASURES

XSS vulnerabilities stand out from other Web attacks by their effects on both the Web application and browser. In the most common scenarios, a Web site must be compromised to serve as the distribution point for the payload. The Web browser then falls victim to the offending code. This implies that countermeasures can be implemented for servers and browsers alike.

Only a handful of browsers pass the 1% market share threshold. Users are at the mercy of those vendors (Apple, Google, Microsoft, Mozilla, Opera) to provide in-browser defenses. Many current popular browsers (Safari 4, Chrome Beta, IE 8, Firefox 3.5) contain some measure of anti-XSS capability. Firefox's NoScript plug-in (<http://noscript.net/>) is of particular note, although it can quickly become an exercise in configuration management. More focus will be given to browser security in Chapter 7, "Web of Distrust."

Preventing XSS is best performed in the Web application itself. The complexities of HTML, JavaScript, and international language support make this a challenging prospect even for security-aware developers.

Fixing a Static Character Set

Character encoding and decoding is prone to error without the added concern of malicious content. A character set should be explicitly set for any of the site's pages that will present dynamic content. This is done either with the Content-Type header or with the HTML meta element via `http-equiv` attribute.

The choice of character set can be influenced by the site's written language, user population, and library support. Some examples from popular Web sites are shown in Table 1.2.

A final hint on using meta elements to set the character set. In the face of vagaries, browsers use MIME-type content sniffing to determine the character set and type of a file. The HTML5 draft specification recommends looking into the first 512 bytes of a file to find, for example, a character set definition. HTML4 provides no guidance, leaving browsers that currently vary between looking at the first 256 to 1,024 bytes.

A corollary to this normalization step is that the declared content type for all user-supplied content should be as explicit as possible. If a Web site expects users to upload image files, in addition to ensuring the files are in fact images of the correct format, the site should serve the images with a correct Content-Type header.

Web site	Character set
www.apple.com	Content-Type: text/html; charset=utf-8
www.baidu.com	Content-Type: text/html; charset=GB2312
www.bing.com	Content-Type: text/html; charset=utf-8
http://news.chinatimes.com	Content-Type: text/html; charset=big5
www.google.com	Content-Type: text/html; charset=ISO-8859-1
www.koora.com	Content-Type: text/html; charset=windows-1256
www.mail.ru	Content-Type: text/html; charset=windows-1251
www.rakuten.co.jp	Content-Type: text/html; charset=x-euc-jp
www.tapuz.co.il	Content-Type: text/html; charset=windows-1255
www.yahoo.com	Content-Type: text/html; charset=utf-8

Normalizing Character Sets and Encoding

A common class of vulnerabilities is called the Race Condition. Race conditions occur when the value of a sensitive token (perhaps a security context identifier or a temporary file) can change between the time its validity is checked and when the value it refers to is used. This is often referred to as a Time of Check, Time of Use (TOCTTOU or TOCTOU) vulnerability. At the time of writing, the Open Web Application Security Project (OWASP) (a site oriented to Web vulnerabilities) last updated its description of TOCTOU on February 21, 2009. As a reminder that computer security predates social networking and cute cat sites, race conditions were discussed as early as 1974.¹

A problem similar to the concept of TOCTO manifests itself with XSS filters and character sets. The input string might be scanned for malicious characters (time of check), some of the string's characters might be decoded, and then the string might be written to a Web page (time of use). Even if some decoding occurs before the time of check, the Web application or its code might perform additional decoding steps. This is where normalization comes in.

Normalization refers to the process in which an input string is transformed into its simplest representation in a fixed character set. For example, all percent-encoded characters are decoded, multibyte sequences are verified to represent a single glyph, and invalid sequences are dealt with (removed, rejected, or replaced). Using the race condition metaphor, this security process could be considered TONTOCTOU – time of normalization, time of check, time of use.

Normalization needs to be considered for input and output.

Invalid sequences should be rejected. Overlong sequences (a representation that uses more bytes than necessary) should be considered invalid.

For the technically oriented, Unicode normalization should use Normalization Form KC to reduce the chances of success for character-based attacks. This basically means that normalization will produce a byte sequence that most concisely represents the intended string. A detailed description of this process, with excellent visual examples of different normalization steps, is at <http://unicode.org/reports/tr15/>.

More information regarding Unicode and security can be found at www.unicode.org/reports/tr39/.

Encoding the Output

If data from the browser will be echoed in a Web page, then the data should be correctly encoded for its destination in the DOM, either with HTML encoding or percent encoding. This is a separate step from normalizing and establishing a fixed character set. HTML encoding represents a character with an entity reference rather than its explicit character code. Not all characters have an entity reference, but the special characters used in XSS payloads to rewrite the DOM do. The HTML4 specification defines the available entities (www.w3.org/TR/REC-html40/sgml/entities.html). Four of the most common entities are shown in Table 1.3.

Encoding special characters that have the potential to manipulate the DOM goes a long way toward preventing XSS attacks.

```
<script>alert("Not encoded")</script>
<script>alert("Encoded")</script>
<input type=text name=search value="living dead" onmouseover=
  alert(/Not encoded/.source)><a href="">
<input type=text name=search value="living dead" onmouseover=
  alert(/Not encoded/.source)<a href="">
```

A similar benefit is gained from using percent encoding when data from the client are to be written in an href attribute or similar. Encoding the quote character as %22 renders it innocuous while preserving its meaning for links. This often occurs, for example, in redirect links.

Different destinations require different encoding steps to preserve the sense of the data. The most common output areas are listed below:

- HTTP headers (such as a Location or Referer), although the exploitability of these locations is difficult if not impossible in many scenarios
- A text node within an element, such as “Welcome to the Machine” between div tags
- An element’s attribute, such as an href, src, or value attribute
- Style properties, such as some ways that a site might enable a user to “skin” the look and feel
- JavaScript variables

Table 1.3 Entity encoding for special characters

Entity encoding	Displayed character
<	<
>	>
&	&
"	"

Review the characters in each area that carry special meaning. For example, if an attribute is enclosed in double quotes, then any user-supplied data to be inserted into that attribute should not contain a double quote or have the quote encoded.

TIP

Any content from the client (whether a header value from the Web browser or text provided by the user) should only be written to the Web page with one or two custom functions, depending on the output location. Regardless of the programming language used by the Web application, replace the language's built-in functions, such as *echo*, *print*, and *writeln*, with a function designed for writing untrusted content to the page with correct encoding for special characters. This makes developers think about the content being displayed to a page and helps a code review identify areas that were missed or may be prone to mistakes.

Beware of Exclusion Lists and Regexes

“Some people, when confronted with a problem, think ‘I know, I’ll use regular expressions’. Now they have two problems.”²

Solely relying on an exclusion list invites application doom. Exclusion lists need to be maintained to deal with changing attack vectors and encoding methods.

Regular expressions are a powerful tool whose complexity is both benefit and curse. Not only might regexes be overly relied upon as a security measure but they are also easily misapplied and misunderstood. A famous regular expression to accurately match the e-mail address format defined in RFC 2822 contains 426 characters (www.regular-expressions.info/email.html). Anyone who would actually take the time to fully understand that regex either would be driven to Lovecraftian insanity or has a strange affinity for mental abuse. Of course, obtaining a near-100% match can be accomplished with much fewer characters. Now, consider these two points: (1) vulnerabilities occur when security mechanisms are inadequate or have mistakes that make them “near-100%” instead of 100% solutions, and (2) regular expressions make poor parsers for even moderately simple syntax.

Fortunately, most user input is expected to fall into somewhat clear categories. The catchword here is “somewhat.” Regular expressions are very good at matching characters within a string but become much more cumbersome when used to match characters or sequences that should not be in a string.

Now that you’ve been warned against placing too much trust in regular expressions, here are some guidelines for using them successfully:

- Work with a normalized character string. Decode HTML-encoded and percent-encoded characters where appropriate.
- Apply the regex at security boundaries – areas where the data will be modified, stored, or rendered to a Web page.
- Work with a character set that the regex engine understands.
- Use a whitelist, or inclusion-based, approach. Match characters that are permitted and reject strings when nonpermitted characters are present.

- Match the entire input string boundaries with the `^` and `$` anchors.
- Reject invalid data; don't try to rewrite it by guessing what characters should be removed.
- If invalid data are to be removed from the input, recursively apply the filter and be fully aware of how the input will be transformed by this removal. If you expect that stripping "script" from all input will prevent script tags from showing up, test your filter against "<script>".
- Don't rely on blocking payloads used by security scanners for your test cases; attackers don't use those payloads.
- Realize when a parser is better suited for the job, such as dealing with HTML elements with attributes or JavaScript.

Where appropriate, use the perlre whitespace prefix, `(?x)`, to make patterns more legible. (This is equivalent to the `PCRE_EXTENDED` option flag in the PCRE library and the `mod_x` syntax option in the Boost.Regex library. Both libraries accept `(?x)` in a pattern.) This causes unescaped whitespace in a pattern to be ignored, thereby giving the creator more flexibility to make the pattern visually understandable by a human.

EPIC FAIL

In August 2009, an XSS vulnerability was revealed in Twitter's application program interface (API). Victims merely needed to view a payload-laden tweet for their browser to be compromised. The discoverer, James Slater, provided an innocuous proof of concept. Twitter quickly responded with a fix. Then the fix was hacked. (www.davidnaylor.co.uk/massive-twitter-cross-site-scripting-vulnerability.html)

The fix? Blacklist spaces from the input – a feat trivially accomplished by a regular expression or even native functions in many programming languages. Clearly, lack of space characters is not an impediment to XSS exploits. Not only did the blacklist approach fail but the first solution demonstrated a lack of understanding of the problem space of defeating XSS attacks.

Reuse, Don't Reimplement, Code

Crypto is the ultimate example of the danger of implementing an algorithm from scratch. Yet the admonition, "Don't create your own crypto," seems to be as effective as "Let's split up" when skulking through a spooky house on a dare.

Frameworks are another example where code reuse is better than writing from scratch. Several JavaScript frameworks were listed in the JSON section. Popular Web languages, such as Java, .NET, PHP, Perl, Python, and Ruby, all have libraries that handle various aspects of Web development.

Of course, reusing insecure code is no better than writing insecure code from scratch. The benefit of JavaScript frameworks is that the chance for programmer mistakes is either reduced or moved to a different location in the application – usually business logic. See Chapter 6, "Logic Attacks," for examples of exploiting the business logic of a Web site.

Microsoft's .NET Anti-XSS library (www.microsoft.com/downloads/details.aspx?FamilyId=051ee83c-5ccf-48ed-8463-02f56a6bfc09&displaylang=en) and the OWASP AntiSamy (www.owasp.org/index.php/Category:OWASP_AntiSamy_Project) project are two examples of security-specific frameworks. Conveniently for this chapter, they provide defenses against XSS attacks.

JavaScript Sandboxes

After presenting an entire chapter on the dangers inherent to running untrusted JavaScript, it would seem bizarre that Web sites would so strongly embrace that very thing. Large Web sites want to tackle the problem of attracting and keeping users. Security, though important, will not be an impediment to innovation when money is on the line.

Web sites compete with each other to offer more dynamic content and offer APIs to develop third-party “weblets” or small browser-based applications that fit within the main site. Third-party applications are a smart way to attract more users and developers to a Web site, turning the site itself into a platform for collecting information and, in the end, making money in one of the few reliable manners – selling and advertising.

The basic approach to a sandbox is to execute the untrusted code within a namespace that might be allowed to access JavaScript functions of a certain site, but otherwise execute in a closed environment. It's very much like the model iPhone uses for its applications or the venerable Java implemented years ago.

Wary developers and weary Web security auditors can find general information about JavaScript and browser security at the Caplet group: <http://tech.groups.yahoo.com/group/caplet/>.

ADsafe (www.adsafe.org/) is designed to protect a site that may be hosting malicious third-party code such as advertising banners or JavaScript widgets. However, its capabilities do not match other, more mature projects.

Caja

Google's approach to in-app sandboxing relies on Caja. Caja uses a capability model to enforce security for untrusted JavaScript. The name plays on the Spanish word for box to create the acronym: capabilities attenuate JavaScript authority. Its major changes to the JavaScript execution environment include immutable objects, reduction of the global environment to a specific code module, and restricted access to sensitive objects and functions.

Caja builds a sandbox around the set of HTML, CSS, and JavaScript that defines some type of functionality – a widget that might display the current weather, stock prices, checking account balances, and so on. The process of creating a sandbox around untrusted code is called *cajoling*. Content goes into the Java-based Caja tool and comes out as JavaScript file that represents the original content as a single *module function*. This translation removes unexpected, unknown, and unsafe content.

Caja is hosted at <http://code.google.com/p/google-caja/>.

Facebook JavaScript

Facebook opened up its site for third-party developers to host their JavaScript/CSS/HTML-based applications directly on the Facebook. These applications would not only be served from a Facebook domain but also be able to interact with users' profiles and friends. Unrestrained JavaScript would wreak havoc across the site. So, Facebook JavaScript (FBJS) was created to encapsulate these potentially dangerous third-party applications in a virtual function scope. It also creates a JavaScript-like environment with reduced functionality so that the hosted applications do not attack the site or each other.

FBJS is hosted at <http://wiki.developers.facebook.com/index.php/FBJS>.

NOTE

An entire chapter on the dangers of XSS and no mention of the browser's same origin policy? This policy defines certain restrictions on the interaction between the DOM and JavaScript. Same origin policy mitigates some ways that XSS vulnerabilities can be exploited, but it has no bearing on the fundamental problem of XSS. In fact, most of the time, the compromised site is serving the payload – placing the attack squarely within the permitted zone of the same origin policy.

SUMMARY

XSS is an ideal exploit venue for attackers across the spectrum of sophistication and programming knowledge. Attack code is easy to write, requiring no more than a text editor and a cursory understanding of JavaScript, unlike buffer overflows. XSS also offers the path of least resistance for a payload that can affect Windows, OSX, Linux, Internet Explorer, Safari, and Opera alike. The Web browser is a universal platform for displaying HTML and interacting with complex Web sites. When that HTML is subtly manipulated by a few malicious characters, the browser becomes a universal platform for exposure.

XSS affects security-aware users whose computers have the latest firewalls, anti-virus software, and security patches installed almost as easily as the casual user, taking a brief moment in a cafe to check e-mail. Successful attacks target data already in the victim's browser or use HTML and JavaScript to force the browser to perform an untoward action. HTML and JavaScript are working behind the scenes inside the browser every time you visit a Web page. From a search engine to Web-based e-mail to reading the news – how often do you inspect every line of text being loaded into the browser?

Some measure of protection can be gained by maintaining an up-to-date browser. The major Web browser vendors continue to add in-browser defenses against the most common forms of XSS and other Web-based exploits. The primary line of defense lays within the Web sites themselves, which must filter, encode, and display content properly to protect visitors from being targeted with XSS.

Endnotes

1. Abbott RP, Chin JS, Donnelley JE, Konigs-Ford WL, Tokubo S, Webb DA. Security analysis and enhancements of computer operating systems. NBSIR 76-1041, National Bureau of Standards, ICST, Washington, D.C.; 1976, p. 19.
2. Zawinski J (an early Netscape Navigator developer repurposing a Unix sed quote), <http://regex.info/blog/2006-09-15/247#comment-3085>; 2006.