the Tangled Web

A Guide to Securing Modern Web Applications

Michal Zalewski

no starch press

CONTENTS IN DETAIL

3

HYPERTEXT TRANSFER PROTOCOL

The next essential concept we need to discuss is the Hypertext Transfer Protocol (HTTP): the core transfer mechanism of the Web and the preferred method for exchanging URL-referenced documents between servers and clients. Despite having hypertext in its name, HTTP and the actual hypertext content (the HTML language) often exist independent of each other. That said, they are intertwined in sometimes surprising ways.

The history of HTTP offers interesting insight into its authors' ambitions and the growing relevance of the Internet. Tim Berners-Lee's earliest 1991 draft of the protocol (HTTP/0.9[1]) was barely one and a half pages long, and it failed to account for even the most intuitive future needs, such as extensibility needed to transmit non-HTML data.

Five years and several iterations of the specification later, the first official HTTP/1.0 standard (RFC 1945[2]) tried to rectify many of these shortcomings in about 50 densely packed pages of text. Fast-forward to 1999, and in HTTP/1.1 (RFC 2616[3]), the seven credited authors attempted to anticipate almost every possible use of the protocol, creating an opus over 150 pages long. That's not all: As of this writing, the current work on HTTPbis,[4] essentially a replacement for the HTTP/1.1 specification, comes to 360 pages or so. While much of the gradually accumulated content is irrelevant to the modern Web, this progression makes it clear that the desire to tack on new features far outweighs the desire to prune failed ones.

Today, all clients and servers support a not-entirely-accurate superset of HTTP/1.0, and most can speak a reasonably complete dialect of HTTP/1.1, with a couple of extensions bolted on. Despite the fact that there is no practical need to do so, several web servers, and all common browsers, also maintain backward compatibility with HTTP/0.9.

Basic Syntax of HTTP Traffic

At a glance, HTTP is a fairly simple, text-based protocol built on top of TCP/IP.* Every HTTP session is initiated by establishing a TCP connection to the server, typically to port 80, and then issuing a request that outlines the requested URL. In response, the server returns the requested file and, in the most rudimentary use case, tears down the TCP connection immediately thereafter.

The original HTTP/0.9 protocol provided no room for any additional metadata to be exchanged between the participating parties. The client request always consisted of a single line, starting with GET, followed by the URL path and query string, and ending with a single CRLF newline (ASCII characters 0x0D 0x0A; servers were also advised to accept a lone LF). A sample HTTP/0.9 request might have looked like this:

```
GET /fuzzy_bunnies.txt
```

In response to this message, the server would have immediately returned the appropriate HTML payload. (The specification required servers to wrap lines of the returned document at 80 characters, but this advice wasn't really followed.)

The HTTP/0.9 approach has a number of substantial deficiencies. For example, it offers no way for browsers to communicate users' language preferences, supply a list of supported document types, and so on. It also gives servers no way to tell a client that the requested file could not be found, that it has moved to a different location, or that the returned file is not an HTML

* *Transmission Control Protocol (TCP)* is one of the core communications protocols of the Internet, providing the transport layer to any application protocols built on top of it. TCP offers reasonably reliable, peer-acknowledged, ordered, session-based connectivity between networked hosts. In most cases, the protocol is also fairly resilient against blind packet spoofing attacks attempted by other, nonlocal hosts on the Internet.

document to begin with. Finally, the scheme is not kind to server administrators: When the transmitted URL information is limited to only the path and query strings, it is impossible for a server to host multiple websites, distinguished by their hostnames, under one IP address—and unlike DNS records, IP addresses don't come cheap.

In order to fix these shortcomings (and to make room for future tweaks), HTTP/1.0 and HTTP/1.1 standards embrace a slightly different conversation format: The first line of a request is modified to include protocol version information, and it is followed by zero or more *name: value* pairs (also known as *headers*), each occupying a separate line. Common request headers included in such requests are *User-Agent* (browser version information), *Host* (URL hostname), *Accept* (supported MIME document types*), *Accept-Language* (supported language codes), and *Referer* (a misspelled field indicating the originating page for the request, if known).

These headers are terminated with a single empty line, which may be followed by any payload the client wishes to pass to the server (the length of which must be explicitly specified with an additional *Content-Length* header). The contents of the payload are opaque from the perspective of the protocol itself; in HTML, this location is commonly used for submitting form data in one of several possible formats, though this is in no way a requirement.

Overall, a simple HTTP/1.1 request may look like this:

```
POST /fuzzy_bunnies/bunny_dispenser.php HTTP/1.1
Host: www.fuzzybunnies.com
User-Agent: Bunny-Browser/1.7
Content-Type: text/plain
Content-Length: 17
Referer: http://www.fuzzybunnies.com/main.html

I REQUEST A BUNNY
```

The server is expected to respond to this query by opening with a line that specifies the supported protocol version, a numerical status code (used to indicate error conditions and otherspecial circumstances), and an optional, human-readable status message. A set of self-explanatory headers comes next, ending with an empty line. The response continues with the contents of the requested resource:

```
HTTP/1.1 200 OK
Server: Bunny-Server/0.9.2
Content-Type: text/plain
Connection: close

BUNNY WISH HAS BEEN GRANTED
```

* MIME type (aka *Internet media type*) is a simple, two-component value identifying the class and format of any given computer file. The concept originated in RFC 2045 and RFC 2046, where it served as a way to describe email attachments. The registry of official values (such as *text/plain* or *audio/mpeg*) is currently maintained by IANA, but ad hoc types are fairly common.

RFC 2616 also permits the response to be compressed in transit using one of three supported methods (*gzip*, *compress*, *deflate*), unless the client explicitly opts out by providing a suitable *Accept-Encoding* header.

The Consequences of Supporting HTTP/0.9

Despite the improvements made in HTTP/1.0 and HTTP/1.1, the unwelcome legacy of the "dumb" HTTP/0.9 protocol lives on, even if it is normally hidden from view. The specification for HTTP/1.0 is partly to blame for this, because it requested that all future HTTP clients and servers support the original, half-baked draft. Specifically, section 3.1 says:

> HTTP/1.0 clients must . . . understand any valid response in the format of HTTP/0.9 or HTTP/1.0.

In later years, RFC 2616 attempted to backtrack on this requirement (section 19.6: "It is beyond the scope of a protocol specification to mandate compliance with previous versions."), but acting on the earlier advice, all modern browsers continue to support the legacy protocol as well.

To understand why this pattern is dangerous, recall that HTTP/0.9 servers reply with nothing but the requested file. There is no indication that the responding party actually understands HTTP and wishes to serve an HTML document. With this in mind, let's analyze what happens if the browser sends an HTTP/1.1 request to an unsuspecting SMTP service running on port 25 of *example.com*:

```
GET /<html><body><h1>Hi! HTTP/1.1
Host: example.com:25
...
```

Because the SMTP server doesn't understand what is going on, it's likely to respond this way:

```
220 example.com ESMTP
500 5.5.1 Invalid command: "GET /<html><body><h1>Hi! HTTP/1.1"
500 5.1.1 Invalid command: "Host: example.com:25"
...
421 4.4.1 Timeout
```

All browsers willing to follow the RFC are compelled to accept these messages as the body of a valid HTTP/0.9 response and assume that the returned document is, indeed, HTML. These browsers will interpret the quoted attacker-controlled snippet appearing in one of the error messages as if it comes from the owners of a legitimate website at *example.com*. This profoundly interferes with the browser security model discussed in Part II of this book and, therefore, is pretty bad.

Newline Handling Quirks

Setting aside the radical changes between HTTP/0.9 and HTTP/1.0, several other core syntax tweaks were made later in the game. Perhaps most notably, contrary to the letter of earlier iterations, HTTP/1.1 asks clients not only to honor newlines in the CRLF and LF format but also to recognize a lone CR character. Although this recommendation is disregarded by the two most popular web servers (IIS and Apache), it is followed on the client side by all browsers except Firefox.

The resulting inconsistency makes it easier for application developers to forget that not only LF but also CR characters must be stripped from any attacker-controlled values that appear anywhere in HTTP headers. To illustrate the problem, consider the following server response, where a user-supplied and insufficiently sanitized value appears in one of the headers, as highlighted in bold:

```
HTTP/1.1 200 OK[CR][LF]
Set-Cookie: last_search_term=[CR][CR]<html><body><h1>Hi![CR][LF]
[CR][LF]
Action completed.
```

To Internet Explorer, this response may appear as:

```
HTTP/1.1 200 OK
Set-Cookie: last_search_term=

<html><body><h1>Hi!

Action completed.
```

In fact, the class of vulnerabilities related to HTTP header newline smuggling—be it due to this inconsistency or just due to a failure to filter any type of a newline—is common enough to have its own name: *header injection* or *response splitting*.

Another little-known and potentially security-relevant tweak is support for multiline headers, a change introduced in HTTP/1.1. According to the standard, any header line that begins with a whitespace is treated as a continuation of the previous one. For example:

```
X-Random-Comment: This is a very long string,
  so why not wrap it neatly?
```

Multiline headers are recognized in client-issued requests by IIS and Apache, but they are not supported by Internet Explorer, Safari, or Opera. Therefore, any implementation that relies on or simply permits this syntax in any attacker-influenced setting may be in trouble. Thankfully, this is rare.

Proxy Requests

Proxies are used by many organizations and Internet service providers to intercept, inspect, and forward HTTP requests on behalf of their users. This may be done to improve performance (by allowing certain server responses to be cached on a nearby system), to enforce network usage policies (for example, to prevent access to porn), or to offer monitored and authenticated access to otherwise separated network environments.

Conventional HTTP proxies depend on explicit browser support: The application needs to be configured to make a modified request to the proxy system, instead of attempting to talk to the intended destination. To request an HTTP resource through such a proxy, the browser will normally send a request like this:

```
GET http://www.fuzzybunnies.com/ HTTP/1.1
User-Agent: Bunny-Browser/1.7
Host: www.fuzzybunnies.com
...
```

The key difference between the above example and the usual syntax is the presence of a fully qualified URL in the first line of the request (*http://www.fuzzybunnies.com/*), instructing the proxy where to connect to on behalf of the user. This information is somewhat redundant, given that the *Host* header already specifies the hostname; the only reason for this overlap is that the mechanisms evolved independent of each other. To avoid being fooled by co-conspiring clients and servers, proxies should either correct any mismatching *Host* headers to match the request URL or associate cached content with a particular URL-*Host* pair and not just one of these values.

Many HTTP proxies also allow browsers to request non-HTTP resources, such as FTP files or directories. In these cases, the proxy will wrap the response in HTTP, and perhaps convert it to HTML if appropriate, before returning it to the user.* That said, if the proxy does not understand the requested protocol, or if it is simply inappropriate for it to peek into the exchanged data (for example, inside encrypted sessions), a different approach must be used. A special type of a request, CONNECT, is reserved for this purpose but is not further explained in the HTTP/1.1 RFC. The relevant request syntax is instead outlined in a separate, draft-only specification from 1998.[5] It looks like this:

```
CONNECT www.fuzzybunnies.com:1234 HTTP/1.1
User-Agent: Bunny-Browser/1.7
...
```

* In this case, some HTTP headers supplied by the client may be used internally by the proxy, but they will not be transmitted to the non-HTTP endpoint, which creates some interesting, if non-security-relevant, protocol ambiguities.

If the proxy is willing and able to connect to the requested destination, it acknowledges this request with a specific HTTP response code, and the role of this protocol ends. At that point, the browser will begin sending and receiving raw binary data within the established TCP stream; the proxy, in turn, is expected to forward the traffic between the two endpoints indiscriminately.

NOTE *Hilariously, due to a subtle omission in the draft spec, many browsers have incorrectly processed the nonencrypted, proxy-originating error responses returned during an attempt to establish an encrypted connection. The affected implementations interpreted such plaintext responses as though they originated from the destination server over a secure channel. This glitch effectively eliminated all assurances associated with the use of encrypted communications on the Web. It took over a decade to spot and correct the flaw.*[6]

Several other classes of lower-level proxies do not use HTTP to communicate directly with the browser but nevertheless inspect the exchanged HTTP messages to cache content or enforce certain rules. The canonical example of this is a transparent proxy that silently intercepts traffic at the TCP/IP level. The approach taken by transparent proxies is unusually dangerous: Any such proxy can look at the destination IP and the *Host* header sent in the intercepted connection, but it has no way of immediately telling if that destination IP is genuinely associated with the specified server name. Unless an additional lookup and correlation is performed, co-conspiring clients and servers can have a field day with this behavior. Without these additional checks, the attacker simply needs to connect to his or her home server and send a misleading *Host: www.google.com* header to have the response cached for all other users as though genuinely coming from *www.google.com.*

Resolution of Duplicate or Conflicting Headers

Despite being relatively verbose, RFC 2616 does a poor job of explaining how a compliant parser should resolve potential ambiguities and conflicts in the request or response data. Section 19.2 of this RFC ("Tolerant Applications") recommends relaxed and error-tolerant parsing of certain fields in "unambiguous" cases, but the meaning of the term itself is, shall we say, not particularly unambiguous.

For example, because of a lack of specification-level advice, roughly half of all browsers will favor the first occurrence of a particular HTTP header, and the rest will favor the last one, ensuring that almost every header injection vulnerability, no matter how constrained, is exploitable for at least some percentage of targeted users. On the server side, the situation is similarly random: Apache will honor the first *Host* header seen, while IIS will completely reject a request with multiple instances of this field.

On a related note, the relevant RFCs contain no explicit prohibition on mixing potentially conflicting HTTP/1.0 and HTTP/1.1 headers and no requirement for HTTP/1.0 servers or clients to ignore all HTTP/1.1 syntax. Because of this design, it is difficult to predict the outcome of indirect conflicts between HTTP/1.0 and HTTP/1.1 directives that are responsible for the same thing, such as *Expires* and *Cache-Control*.

Finally, in some rare cases, header conflict resolution is outlined in the spec very clearly, but the purpose of permitting such conflicts to arise in the first place is much harder to understand. For example, HTTP/1.1 clients are required to send the *Host* header on all requests, but servers (not just proxies!) are also required to recognize absolute URLs in the first line of the request, as opposed to the traditional path- and query-only method. This rule permits a curiosity such as this:

```
GET http://www.fuzzybunnies.com/ HTTP/1.1
Host: www.bunnyoutlet.com
```

In this case, section 5.2 of RFC 2616 instructs clients to disregard the nonfunctional (but still mandatory!) *Host* header, and many implementations follow this advice. The problem is that underlying applications are likely to be unaware of this quirk and may instead make somewhat important decisions based on the inspected header value.

NOTE *When complaining about the omissions in the HTTP RFCs, it is important to recognize that the alternatives can be just as problematic. In several scenarios outlined in that RFC, the desire to explicitly mandate the handling of certain corner cases led to patently absurd outcomes. One such example is the advice on parsing dates in certain HTTP headers, at the request of section 3.3 in RFC 1945. The resulting implementation (the* prtime.c *file in the Firefox codebase*[7]*) consists of close to 2,000 lines of extremely confusing and unreadable C code just to decipher the specified date, time, and time zone in a sufficiently fault-tolerant way (for uses such as deciding cache content expiration).*

Semicolon-Delimited Header Values

Several HTTP headers, such as *Cache-Control* or *Content-Disposition*, use a semicolon-delimited syntax to cram several separate *name=value* pairs into a single line. The reason for allowing this nested notation is unclear, but it is probably driven by the belief that it will be a more efficient or a more intuitive approach that using several separate headers that would always have to go hand in hand.

Some use cases outlined in RFC 2616 permit *quoted-string* as the right-hand parameter in such pairs. *Quoted-string* is a syntax in which a sequence of arbitrary printable characters is surrounded by double quotes, which act as delimiters. Naturally, the quote mark itself cannot appear inside the string, but—importantly—a semicolon or a whitespace may, permitting many otherwise problematic values to be sent as is.

Unfortunately for developers, Internet Explorer does not cope with the *quoted-string* syntax particularly well, effectively rendering this encoding scheme useless. The browser will parse the following line (which is meant to indicate that the response is a downloadable file rather than an inline document) in an unexpected way:

```
Content-Disposition: attachment; filename="evil_file.exe;.txt"
```

In Microsoft's implementation, the filename will be truncated at the semicolon character and will appear to be *evil_file.exe*. This behavior creates a potential hazard to any application that relies on examining or appending a "safe" filename extension to an attacker-controlled filename and otherwise correctly checks for the quote character and newlines in this string.

NOTE *An additional* quoted-pair *mechanism is provided to allow quotes (and any other characters) to be used safely in the string when prefixed by a backslash. This mechanism appears to be specified incorrectly, however, and not supported by any major browser except for Opera. For* quoted-pair *to work properly, stray "\" characters would need to be banned from the* quoted-string, *which isn't the case in RFC 2616.* Quoted-pair *also permits any* CHAR-*type token to be quoted, including newlines, which is incompatible with other HTTP-parsing rules.*

It is also worth noting that when duplicate semicolon-delimited fields are found in a single HTTP header, their order of precedence is not defined by the RFC. In the case of *filename=* in *Content-Disposition*, all mainstream browsers use the first occurrence. But there is little consistency elsewhere. For example, when extracting the *URL=* value from the *Refresh* header (used to force reloading the page after a specified amount of time), Internet Explorer 6 will fall back to the last instance, yet all other browsers will prefer the first one. And when handling *Content-Type*, Internet Explorer, Safari, and Opera will use the first *charset=* value, while Firefox and Chrome will rely on the last.

NOTE *Food for thought: A fascinating but largely non-security-related survey of dozens of inconsistencies associated with the handling of just a single HTTP header—* Content-Disposition*—can be found on a page maintained by Julian Reschke:* http://greenbytes.de/tech/tc2231/.

Header Character Set and Encoding Schemes

Like the documents that laid the groundwork for URL handling, all subsequent HTTP specs have largely avoided the topic of dealing with non-US-ASCII characters inside header values. There are several plausible scenarios where non-English text may legitimately appear in this context (for example, the filename in *Content-Disposition*), but when it comes to this, the expected browser behavior is essentially undefined.

Originally, RFC 1945 permitted the TEXT token (a primitive broadly used to define the syntax of other fields) to contain 8-bit characters, providing the following definition:

```
OCTET          = <any 8-bit sequence of data>
CTL            = <any US-ASCII control character
                  (octets 0 - 31) and DEL (127)>
TEXT           = <any OCTET except CTLs,
                  but including LWS>
```

The RFC followed up with cryptic advice: When non-US-ASCII characters are encountered in a TEXT field, clients and servers *may* interpret them as ISO-8859-1, the standard Western European code page, but they don't have to. Later, RFC 2616 copied and pasted the same specification of TEXT tokens but added a note that non-ISO-8859-1 strings must be encoded using a format outlined in RFC 2047,[8] originally created for email communications. Fair enough; in this simple scheme, the encoded string opens with a "=?" prefix, followed by a character-set name,a "?q?" or "?b?" encoding-type indicator (*quoted-printable*[*] or *base64*,[†] respectively), and lastly the encoded string itself. The sequence ends with a "?=" terminator. An example of this may be:

```
Content-Disposition: attachment; filename="=?utf-8?q?Hi=21.txt?="
```

NOTE *The RFC should also have stated that any spurious "=?... ?=" patterns must never be allowed as is in the relevant headers, in order to avoid unintended decoding of values that were not really encoded to begin with.*

Sadly, the support for this RFC 2047 encoding is spotty. It is recognized in some headers by Firefox and Chrome, but other browsers are less cooperative. Internet Explorer chooses to recognize URL-style percent encoding in the *Content-Disposition* field instead (a habit also picked up by Chrome) and defaults to UTF-8 in this case. Firefox and Opera, on the other hand, prefer supporting a peculiar percent-encoded syntax proposed in RFC 2231,[9] a striking deviation from how HTTP syntax is supposed to look:

```
Content-Disposition: attachment; filename*=utf-8'en-us'Hi%21.txt
```

Astute readers may notice that there is no single encoding scheme supported by all browsers at once. This situation prompts some web application developers to resort to using raw high-bit values in the HTTP headers, typically interpreted as UTF-8, but doing so is somewhat unsafe. In Firefox, for example, a long-standing glitch causes UTF-8 text to be mangled when put

[*] *Quoted-printable* is a simple encoding scheme that replaces any nonprintable or otherwise illegal characters with the equal sign (=) followed by a 2-digit hexadecimal representation of the 8-bit character value to be encoded. Any stray equal signs in the input text must be replaced with "=3D" as well.

[†] *Base64* is a non-human-readable encoding that encodes arbitrary 8-bit input using a 6-bit alphabet of case-sensitive alphanumerics, "+", and "/". Every 3 bytes of input map to 4 bytes of output. If the input does not end at a 3-byte boundary, this is indicated by appending one or two equal signs at the end of the output string.

in the *Cookie* header, permitting attacker-injected cookie delimiters to materialize in unexpected places.[10] In other words, there are no easy and robust solutions to this mess.

When discussing character encodings, the problem of handling of the NUL character (0x00) probably deserves a mention. This character, used as a string terminator in many programming languages, is technically prohibited from appearing in HTTP headers (except for the aforementioned, dysfunctional *quoted-pair* syntax), but as you may recall, parsers are encouraged to be tolerant. When this character is allowed to go through, it is likely to have unexpected side effects. For example, *Content-Disposition* headers are truncated at NUL by Internet Explorer, Firefox, and Chrome but not by Opera or Safari.

Referer Header Behavior

As mentioned earlier in this chapter, HTTP requests may include a *Referer* header. This header contains the URL of a document that triggered the current navigation in some way. It is meant to help with certain troubleshooting tasks and to promote the growth of the Web by emphasizing cross-references between related web pages.

Unfortunately, the header may also reveal some information about user browsing habits to certain unfriendly parties, and it may leak sensitive information that is encoded in the URL query parameters on the referring page. Due to these concerns, and the subsequent poor advice on how to mitigate them, the header is often misused for security or policy enforcement purposes, but it is not up to the task. The main problem is that there is no way to differentiate between a client that is not providing the header because of user privacy preferences, one that is not providing it because of the type of navigation taking place, and one that is deliberately tricked into hiding this information by a malicious referring site.

Normally, this header is included in most HTTP requests (and preserved across HTTP-level redirects), except in the following scenarios:

- After organically entering a new URL into the address bar or opening a bookmarked page.
- When the navigation originates from a pseudo-URL document, such as *data:* or *javascript:*.
- When the request is a result of redirection controlled by the *Refresh* header (but not a *Location*-based one).
- Whenever the referring site is encrypted but the requested page isn't. According to RFC 2616 section 15.1.2, this is done for privacy reasons, but it does not make a lot of sense. The *Referer* string is still disclosed to third parties when one navigates from one encrypted domain to an unrelated encrypted one, and rest assured, the use of encryption is not synonymous with trustworthiness.
- If the user decides to block or spoof the header by tweaking browser settings or installing a privacy-oriented plug-in.

As should be apparent, four out of five of these conditions can be purposefully induced by any rogue site.

HTTP Request Types

The original HTTP/0.9 draft provided a single method (or "verb") for requesting a document: GET. The subsequent proposals experimented with an increasingly bizarre set of methods to permit interactions other than retrieving a document or running a script, including such curiosities as SHOWMETHOD, CHECKOUT, or—why not—SPACEJUMP.[11]

Most of these thought experiments have been abandoned in HTTP/1.1, which settles on a more manageable set of eight methods. Only the first two request types—GET and POST—are of any significance to most of the modern Web.

GET

The GET method is meant to signify information retrieval. In practice, it is used for almost all client-server interactions in the course of a normal browsing session. Regular GET requests carry no browser-supplied payloads, although they are not strictly prohibited from doing so.

The expectation is that GET requests should not have, to quote the RFC, "significance of taking an action other than retrieval" (that is, they should make no persistent changes to the state of the application). This requirement is increasingly meaningless in modern web applications, where the application state is often not even managed entirely on the server side; consequently, the advice is widely ignored by application developers.*

NOTE *In HTTP/1.1, clients may ask the server for any set of possibly noncontiguous or overlapping fragments of the target document by specifying the* Range *header on GET (and, less commonly, on some other types of requests). The server is not obliged to comply, but where the mechanism is available, browsers may use it to resume aborted downloads.*

POST

The POST method is meant for submitting information (chiefly HTML forms) to the server for processing. Because POST actions may have persistent side effects, many browsers ask the user to confirm before reloading any content retrieved with POST, but for the most part, GET and POST are used in a quasi-interchangeable manner.

POST requests are commonly accompanied by a payload, the length of which is indicated by the *Content-Length* header. In the case of plain HTML, the payload may consist of URL-encoded or MIME-encoded form data (a format detailed in Chapter 4), although again, the syntax is not constrained at the HTTP level in any special way.

* There is an anecdotal (and perhaps even true) tale of an unfortunate webmaster by the name of John Breckman. According to the story, John's website has been accidentally deleted by a search engine–indexing robot. The robot simply unwittingly discovered an unauthenticated, GET-based administrative interface that John had built for his site . . . and happily followed every "delete" link it could find.

HEAD

HEAD is a rarely used request type that is essentially identical to GET but that returns only the HTTP headers, and not the actual payload, for the requested content. Browsers generally do not issue HEAD requests on their own, but the method is sometimes employed by search engine bots and other automated tools, for example, to probe for the existence of a file or to check its modification time.

OPTIONS

OPTIONS is a metarequest that returns the set of supported methods for a particular URL (or "*", meaning the server in general) in a response header. The OPTIONS method is almost never used in practice, except for server fingerprinting; because of its limited value, the returned information may not be very accurate.

NOTE *For the sake of completeness, we need to note that OPTIONS requests are also a cornerstone of a proposed cross-domain request authorization scheme, and as such, they may gain some prominence soon. We will revisit this scheme, and explore many other upcoming browser security features, in Chapter 16.*

PUT

A PUT request is meant to allow files to be uploaded to the server at the specified target URL. Because browsers do not support PUT, intentional file-upload capabilities are almost always implemented through POST to a server-side script, rather than with this theoretically more elegant approach.

That said, some nonweb HTTP clients and servers may use PUT for their own purposes. Just as interestingly, some web servers may be misconfigured to process PUT requests indiscriminately, creating an obvious security risk.

DELETE

DELETE is a self-explanatory method that complements PUT (and that is equally uncommon in practice).

TRACE

TRACE is a form of "ping" request that returns information about all the proxy hops involved in processing a request and echoes the original request as well. TRACE requests are not issued by web browsers and are seldom used for legitimate purposes. TRACE's primary use is for security testing, where it may reveal interesting details about the internal architecture of HTTP servers in a remote network. Precisely for this reason, the method is often disabled by server administrators.

CONNECT

The CONNECT method is reserved for establishing non-HTTP connections through HTTP proxies. It is not meant to be issued directly to servers. If the support for CONNECT request is enabled accidentally on a particular server, it may pose a security risk by offering an attacker a way to tunnel TCP traffic into an otherwise protected network.

Other HTTP Methods

A number of other request methods may be employed by other nonbrowser applications or browser extensions; the most popular set of HTTP extensions may be WebDAV, an authoring and version-control protocol described in RFC 4918.[12]

Further, the *XMLHttpRequest* API nominally allows client-side JavaScript to make requests with almost arbitrary methods to the originating server—although this last functionality is heavily restricted in certain browsers (we will look into this in Chapter 9).

Server Response Codes

Section 10 of RFC 2616 lists nearly 50 status codes that a server may choose from when constructing a response. About 15 of these are used in real life, and the rest are used to indicate increasingly bizarre or unlikely states, such as "402 Payment Required" or "415 Unsupported Media Type." Most of the RFC-listed states do not map cleanly to the behavior of modern web applications; the only reason for their existence is that somebody hoped they eventually would.

A few codes are worth memorizing because they are common or carry special meaning, as discussed below.

200–299: Success

This range of status codes is used to indicate a successful completion of a request:

200 OK This is a normal response to a successful GET or POST. The browser will display the subsequently returned payload to the user or will process it in some other context-specific way.

204 No Content This code is sometimes used to indicate a successful request to which no verbose response is expected. A 204 response aborts navigation to the URL that triggered it and keeps the user on the originating page.

206 Partial Content This code is like 200, except that it is returned by servers in response to range requests. The browser must already have a portion of the document (or it would not have issued a range request) and will normally inspect the *Content-Range* response header to reassemble the document before further processing it.

300–399: Redirection and Other Status Messages

These codes are used to communicate a variety of states that do not indicate an error but that require special handling on the browser end:

301 Moved Permanently, 302 Found, 303 See Other This response instructs the browser to retry the request at a new location, specified in the *Location* response header. Despite the distinctions made in the RFC, when encountering any of these response codes, all modern browsers replace POST with GET, remove the payload, and then resubmit the request automatically.

NOTE *Redirect messages may contain a payload, but if they do, this message will not be shown to the user unless the redirection is not possible (for example, because of a missing or unsupported* Location *value). In fact, in some browsers, display of the message may be suppressed even in that scenario.*

304 Not Modified This nonredirect response instructs the client that the requested document hasn't been modified in relation to the copy the client already has. This response is seen after conditional requests with headers such as *If-Modified-Since*, which are issued to revalidate the browser document cache. The response body is not shown to the user. (If the server responds this way to an unconditional request, the result will be browser-specific and may be hilarious; for example, Opera will pop up a nonfunctional download prompt.)

307 Temporary Redirect Similar to 302, but unlike with other modes of redirection, browsers will not downgrade POST to GET when following a 307 redirect. This code is not commonly used in web applications, and some browsers do not behave very consistently when handling it.

400–499: Client-Side Error

This range of codes is used to indicate error conditions caused by the behavior of the client:

400 Bad Request (and related messages) The server is unable or unwilling to process the request for some unspecified reason. The response payload will usually explain the problem to some extent and will be typically handled by the browser just like a 200 response.

More specific variants, such as "411 Length Required," "405 Method Not Allowed," or "414 Request-URI Too Long," also exist. It's anyone's guess as to why not specifying *Content-Length* when required has a dedicated 411 response code but not specifying *Host* deserves only a generic 400 one.

401 Unauthorized This code means that the user needs to provide protocol-level HTTP authentication credentials in order to access the resource. The browser will usually prompt the user for login information next, and it will present a response body only if the authentication process is unsuccessful. This mechanism will be explained in more detail shortly, in "HTTP Authentication" on page 62.

403 Forbidden The requested URL exists but can't be accessed for reasons other than incorrect HTTP authentication. Reasons may involve insufficient filesystem permissions, a configuration rule that prevents this request from being processed, or insufficient credentials of some sort (e.g., invalid cookies or an unrecognized source IP address). The response will usually be shown to the user.

404 Not Found The requested URL does not exist. The response body is typically shown to the user.

500–599: Server-Side Error

This is a class of error messages returned in response to server-side problems:

500 Internal Server Error, 503 Service Unavailable, and so on The server is experiencing a problem that prevents it from fulfilling the request. This may be a transient condition, a result of misconfiguration, or simply the effect of requesting an unexpected location. The response is normally shown to the user.

Consistency of HTTP Code Signaling

Because there is no immediately observable difference between returning most 2xx, 4xx, and 5xx codes, these values are not selected with any special zeal. In particular, web applications are notorious for returning "200 OK" even when an application error has occurred and is communicated on the resulting page. (This is one of the many factors that make automated testing of web applications much harder than it needs to be.)

On rare occasions, new and not necessarily appropriate HTTP codes are invented for specific uses. Some of these are standardized, such as a couple of messages introduced in the WebDAV RFC.[13] Others, such as Microsoft's Microsoft Exchange "449 Retry With" status, are not.

Keepalive Sessions

Originally, HTTP sessions were meant to happen in one shot: Make one request for each TCP connection, rinse, and repeat. The overhead of repeatedly completing a three-step TCP handshake (and forking off a new process in the traditional Unix server design model) soon proved to be a bottleneck, so HTTP/1.1 standardized the idea of keepalive sessions instead.

The existing protocol already gave the server an understanding of where the client request ended (an empty line, optionally followed by *Content-Length* bytes of data), but to continue using the existing connection, the client also needed to know the same about the returned document; the termination of a connection could no longer serve as an indicator. Therefore, keepalive sessions require the response to include a *Content-Length* header too, always specifying the amount of data to follow. Once this many payload bytes are received, the client knows it is okay to send a second request and begin waiting for another response.

Although very beneficial from a performance standpoint, the way this mechanism is designed exacerbates the impact of HTTP request and response-splitting bugs. It is deceptively easy for the client and the server to get out of sync on which response belongs to which request. To illustrate, let's consider a server that thinks it is sending a single HTTP response, structured as follows:

```
HTTP/1.1 200 OK[CR][LF]
Set-Cookie: term=[CR]Content-Length: 0[CR][CR]HTTP/1.1 200 OK[CR]Gotcha: Yup[CR][LF]
Content-Length: 17[CR][LF]
[CR][LF]
Action completed.
```

The client, on the other hand, may see two responses and associate the first one with its most current request and the second one with the yet-to-be-issued query[*] (which may even be addressed to a different hostname on the same IP):

```
HTTP/1.1 200 OK
Set-Cookie: term=
Content-Length: 0

HTTP/1.1 200 OK
Gotcha: Yup
Content-Length: 17

Action completed.
```

If this response is seen by a caching HTTP proxy, the incorrect result may also be cached globally and returned to other users, which is really bad news. A much safer design for keepalive sessions would involve specifying the length of both the headers and the payload up front or using a randomly generated and unpredictable boundary to delimit every response. Regrettably, the design does neither.

Keepalive connections are the default in HTTP/1.1 unless they are explicitly turned off (*Connection: close*) and are supported by many HTTP/1.0 servers when enabled with a *Connection: keep-alive* header. Both servers and browsers can limit the number of concurrent requests serviced per connection and can specify the maximum amount of time an idle connection is kept around.

Chunked Data Transfers

The significant limitation of *Content-Length*-based keepalive sessions is the need for the server to know in advance the exact size of the returned response. This is a pretty simple task when dealing with static files, as the

[*] In principle, clients could be designed to sink any unsolicited server response data before issuing any subsequent requests in a keepalive session, limiting the impact of the attack. This proposal is undermined by the practice of HTTP pipelining, however; for performance reasons, some clients are designed to dump multiple requests at once, without waiting for a complete response in between.

information is already available in the filesystem. When serving dynamically generated data, the problem is more complicated, as the output must be cached in its entirety before it is sent to the client. The challenge becomes insurmountable if the payload is very large or is produced gradually (think live video streaming). In these cases, precaching to compute payload size is simply out of the question.

In response to this challenge, RFC 2616 section 3.6.1 gives servers the ability to use *Transfer-Encoding: chunked*, a scheme in which the payload is sent in portions as it becomes available. The length of every portion of the document is declared up front using a hexadecimal integer occupying a separate line, but the total length of the document is indeterminate until a final zero-length chunk is seen.

A sample chunked response may look like this:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
...

5
Hello
6
world!
0
```

There are no significant downsides to supporting chunked data transfers, other than the possibility of pathologically large chunks causing integer overflows in the browser code or needing to resolve mismatches between *Content-Length* and chunk length. (The specification gives precedence to chunk length, although any attempts to handle this situation gracefully appear to be ill-advised.) All the popular browsers deal with these conditions properly, but new implementations need to watch their backs.

Caching Behavior

For reasons of performance and bandwidth conservation, HTTP clients and some intermediaries are eager to cache HTTP responses for later reuse. This must have seemed like a simple task in the early days of the Web, but it is increasingly fraught with peril as the Web encompasses ever more sensitive, user-specific information and as this information is updated more and more frequently.

RFC 2616 section 13.4 states that GET requests responded to with a range of HTTP codes (most notably, "200 OK" and "301 Moved Permanently") may be implicitly cached in the absence of any other server-provided directives. Such a response may be stored in the cache indefinitely, and may be reused for any future requests involving the same request method and destination URL, even if other parameters (such as *Cookie* headers) differ. There is a prohibition against caching requests that use HTTP authentication (see "HTTP Authentication" on page 62), but other authentication methods, such as cookies, are not recognized in the spec.

When a response is cached, the implementation may opt to revalidate it before reuse, but doing so is not required most of the time. Revalidation is achieved by request with a special conditional header, such as *If-Modified-Since* (followed by a date recorded on the previously cached response) or *If-None-Match* (followed by an opaque *ETag* header value that the server returned with an earlier copy). The server may respond with a "304 Not Modified" code or return a newer copy of the resource.

NOTE *The* Date/If-Modified-Since *and* ETag/If-None-Match *header pairs, when coupled with* Cache-Control: private*, offer a convenient and entirely unintended way for websites to store long-lived, unique tokens in the browser.*[14] *The same can also be achieved by depositing a unique token inside a cacheable JavaScript file and returning "304 Not Modified" to all future conditional requests to the token-generating location. Unlike purpose-built mechanisms such as HTTP cookies (discussed in the next section), users have very little control over what information is stored in the browser cache, under what circumstances, and for how long.*

Implicit caching is highly problematic, and therefore, servers almost always should resort to using explicit HTTP-caching directives. To assist with this, HTTP/1.0 provides an *Expires* header that specifies the date by which the cached copy should be discarded; if this value is equal to the *Date* header provided by the server, the response is noncacheable. Beyond that simple rule, the connection between *Expires* and *Date* is unspecified: It is not clear whether *Expires* should be compared to the system clock on the caching system (which is problematic if the client and server clocks are not in sync) or evaluated based on the *Expires – Date* delta (which is more robust, but which may stop working if *Date* is accidentally omitted). Firefox and Opera use the latter interpretation, while other browsers prefer the former one. In most browsers, an invalid *Expires* value also inhibits caching, but depending on it is a risky bet.

HTTP/1.0 clients can also include a *Pragma: no-cache* request header, which may be interpreted by the proxy as an instruction to obtain a new copy of the requested resource, instead of returning an existing one. Some HTTP/1.0 proxies also recognize a nonstandard *Pragma: no-cache* response header as an instruction not to make a copy of the document.

In contrast, HTTP/1.1 embraces a far more substantial approach to caching directives, introducing a new *Cache-Control* header. The header takes values such as *public* (the document is cacheable publicly), *private* (proxies are not permitted to cache), *no-cache* (which is a bit confusing—the response may be cached but should not be reused for future requests),* and *no-store* (absolutely no caching at all). Public and private caching directives may be accompanied with a qualifier such as *max-age*, specifying the maximum time an old copy should be kept, or *must-revalidate*, requesting a conditional request to be made before content reuse.

* The RFC is a bit hazy in this regard, but it appears that the intent is to permit the cached document to be used for purposes such as operating the "back" and "forward" navigation buttons in a browser but not when a proper page load is requested. Firefox follows this approach, while all other browsers consider *no-cache* and *no-store* to be roughly equivalent.

Unfortunately, it is typically necessary for servers to return both HTTP/1.0 and HTTP/1.1 caching directives, because certain types of legacy commercial proxies do not understand *Cache-Control* correctly. In order to reliably prevent caching over HTTP, it may be necessary to use the following set of response headers:

```
Expires: [current date]
Date: [current date]
Pragma: no-cache
Cache-Control: no-cache, no-store
```

When these caching directives disagree, the behavior is difficult to predict: Some browsers will favor HTTP/1.1 directives and give precedence to *no-cache*, even if it is mistakenly followed by *public*; others don't.

Another risk of HTTP caching is associated with unsafe networks, such as public Wi-Fi networks, which allow an attacker to intercept requests to certain URLs and return modified, long-cacheable contents on requests to the victim. If such a poisoned browser cache is then reused on a trusted network, the injected content will unexpectedly resurface. Perversely, the victim does not even have to visit the targeted application: A reference to a carefully chosen sensitive domain can be injected by the attacker into some other context. There are no good solutions to this problem yet; purging your browser cache after visiting Starbucks may be a very good idea.

HTTP Cookie Semantics

HTTP cookies are not a part of RFC 2616, but they are one of the more important protocol extensions used on the Web. The cookie mechanism allows servers to store short, opaque *name=value* pairs in the browser by sending a *Set-Cookie* response header and to receive them back on future requests via the client-supplied *Cookie* parameter. Cookies are by far the most popular way to maintain sessions and authenticate user requests; they are one of the four canonical forms of *ambient authority*[*] on the Web (the other forms being built-in HTTP authentication, IP checking, and client certificates).

Originally implemented in Netscape by Lou Montulli around 1994, and described in a brief four-page draft document,[15] the mechanism has not been outlined in a proper standard in the last 17 years. In 1997, RFC 2109[16] attempted to document the status quo, but somewhat inexplicably, it also proposed a number of sweeping changes that, to this day, make this specification substantially incompatible with the actual behavior of any modern browser. Another ambitious effort—*Cookie2*—made an appearance in RFC 2965,[17] but a decade later, it still has virtually no browser-level support, a situation that is

[*] *Ambient authority* is a form of access control based on a global and persistent property of the requesting entity, rather than any explicit form of authorization that would be valid only for a specific action. A user-identifying cookie included indiscriminately on every outgoing request to a remote site, without any consideration for why this request is being made, falls into that category.

unlikely to change. A new effort to write a reasonably accurate cookie specification—RFC 6265[18]—was wrapped up shortly before the publication of this book, finally ending this specification-related misery.

Because of the prolonged absence of any real standards, the actual implementations evolved in very interesting and sometimes incompatible ways. In practice, new cookies can be set using *Set-Cookie* headers followed by a single *name=value* pair and a number of optional semicolon-delimited parameters defining the scope and lifetime of the cookie.

Expires Specifies the expiration date for a cookie in a format similar to that used for *Date* or *Expires* HTTP headers. If a cookie is served without an explicit expiration date, it is typically kept in memory for the duration of a browser session (which, especially on portable computers with suspend functionality, can easily span several weeks). Definite-expiry cookies may be routinely saved to disk and persist across sessions, unless a user's privacy settings explicitly prevent this possibility.

Max-age This alternative, RFC-suggested expiration mechanism is not supported in Internet Explorer and therefore is not used in practice.

Domain This parameter allows the cookie to be scoped to a domain broader than the hostname that returned the *Set-Cookie* header. The exact rules and security consequences of this scoping mechanism are explored in Chapter 9.

NOTE *Contrary to what is implied in RFC 2109, it is not possible to scope cookies to a specific hostname when using this parameter. For example,* domain=example.com *will always match* www.example.com *as well. Omitting* domain *is the only way to create host-scoped cookies, but even this approach is not working as expected in Internet Explorer.*

Path Allows the cookie to be scoped to a particular request path prefix. This is not a viable security mechanism for the reasons explained in Chapter 9, but it may be used for convenience, to prevent identically named cookies used in various parts of the application from colliding with each other.

Secure attribute Prevents the resulting cookie from being sent over nonencrypted connections.

HttpOnly attribute Removes the ability to read the cookie through the *document.cookie* API in JavaScript. This is a Microsoft extension, although it is now supported by all mainstream browsers.

When making future requests to a domain for which valid cookies are found in the cookie jar, browsers will combine all applicable *name=value* pairs into a single, semicolon-delimited *Cookie* header, without any additional metadata, and return them to the server. If too many cookies need to be sent on a particular request, server-enforced header size limits will be exceeded, and the request may fail; there is no method for recovering from this condition, other than manually purging the cookie jar.

Curiously, there is no explicit method for HTTP servers to delete unneeded cookies. However, every cookie is uniquely identified by a name-domain-path tuple (the *secure* and *httponly* attributes are ignored), which permits an old cookie of a known scope to be simply overwritten. Furthermore, if the overwriting cookie has an *expires* date in the past, it will be immediately dropped, effectively giving a contrived way to purge the data.

Although RFC 2109 requires multiple comma-separated cookies to be accepted within a single *Set-Cookie* header, this approach is dangerous and is no longer supported by any browser. Firefox allows multiple cookies to be set in a single step via the *document.cookie* JavaScript API, but inexplicably, it requires newlines as delimiters instead. No browser uses commas as *Cookie* delimiters, and recognizing them on the server side should be considered unsafe.

Another important difference between the spec and reality is that cookie values are supposed to use the *quoted-string* format outlined in HTTP specs (see "Semicolon-Delimited Header Values" on page 48), but only Firefox and Opera recognize this syntax in practice. Reliance on *quoted-string* values is therefore unsafe, and so is allowing stray quote characters in attacker-controlled cookies.

Cookies are not guaranteed to be particularly reliable. User agents enforce modest settings on the number and size of cookies permitted per domain and, as a misguided privacy feature, may also restrict their lifetime. Because equally reliable user tracking may be achieved by other means, such as the *ETag/If-None-Match* behavior outlined in the previous section, the efforts to restrict cookie-based tracking probably do more harm than good.

HTTP Authentication

HTTP authentication, as specified in RFC 2617,[19] is the original credential-handling mechanism envisioned for web applications, one that is now almost completely extinct. The reasons for this outcome might have been the inflexibility of the associated browser-level UIs, the difficulty of accommodating more sophisticated non-password-based authentication schemes, or perhaps the inability to exercise control over how long credentials are cached and what other domains they are shared with.

In any case, the basic scheme is fairly simple. It begins with the browser making an unauthenticated request, to which the server responds with a "401 Unauthorized" code.[*] The server must also include a *WWW-Authenticate* HTTP header, specifying the requested authentication method, the *realm* string (an arbitrary identifier to which the entered credentials should be bound), and other method-specific parameters, if applicable.

[*] The terms *authentication* and *authorization* appear to be used interchangeably in this RFC, but they have a distinctive meaning elsewhere in information security. *Authentication* is commonly used to refer to the process of proving your identity, whereas *authorization* is the process of determining whether your previously established credentials permit you to carry out a specific privileged action.

The client is expected to obtain the credentials in one way or the other, encode them in the *Authorization* header, and retry the original request with this header included. According to the specification, for performance reasons, the same *Authorization* header may also be included on subsequent requests to the same server path prefix without the need for a second *WWW-Authenticate* challenge. It is also permissible to reuse the same credentials in response to any *WWW-Authenticate* challenges elsewhere on the server, if the *realm* string and the authentication method match.

In practice, this advice is not followed very closely: Other than Safari and Chrome, most browsers ignore the *realm* string or take a relaxed approach to path matching. On the flip side, all browsers scope cached credentials not only to the destination server but also to a specific protocol and port, a practice that offers some security benefits.

The two credential-passing methods specified in the original RFC are known as *basic* and *digest*. The first one essentially sends the passwords in plaintext, encoded as *base64*. The other computes a one-time cryptographic hash that protects the password from being viewed in plaintext and prevents the *Authorization* header from being replayed later. Unfortunately, modern browsers support both methods and do not distinguish between them in any clear way. As a result, attackers can simply replace the word *digest* with *basic* in the initial request to obtain a clean, plaintext password as soon as the user completes the authentication dialog. Surprisingly, section 4.8 of the RFC predicted this risk and offered some helpful yet ultimately ignored advice:

> User agents should consider measures such as presenting a visual indication at the time of the credentials request of what authentication scheme is to be used, or remembering the strongest authentication scheme ever requested by a server and produce a warning message before using a weaker one. It might also be a good idea for the user agent to be configured to demand Digest authentication in general, or from specific sites.

In addition to these two RFC-specified authentication schemes, some browsers also support less-common methods, such as Microsoft's *NTLM* and *Negotiate*, used for seamless authentication with Windows domain credentials.[20]

Although HTTP authentication is seldom encountered on the Internet, it still casts a long shadow over certain types of web applications. For example, when an external, attacker-supplied image is included in a thread on a message board, and the server hosting that image suddenly decides to return "401 Unauthorized" on some requests, users viewing the thread will be presented out of the blue with a somewhat cryptic password prompt. After double-checking the address bar, many will probably confuse the prompt for a request to enter their forum credentials, and these will be immediately relayed to the attacker's image-hosting server. Oops.

Protocol-Level Encryption and Client Certificates

As should now be evident, all information in HTTP sessions is exchanged in plaintext over the network. In the 1990s, this would not have been a big deal: Sure, plaintext exposed your browsing choices to nosy ISPs, and perhaps to another naughty user on your office network or an overzealous government agency, but that seemed no worse than the behavior of SMTP, DNS, or any other commonly used application protocol. Alas, the growing popularity of the Web as a commerce platform has aggravated the risk, and substantial network security regression caused by the emergence of inherently unsafe public wireless networks put another nail in that coffin.

After several less successful hacks, a straightforward solution to this problem was proposed in RFC 2818:[21] Why not encapsulate normal HTTP requests within an existing, multipurpose Transport Layer Security (TLS, aka SSL) mechanism developed several years earlier? This transport method leverages public key cryptography[*] to establish a confidential, authenticated communication channel between the two endpoints, without requiring any HTTP-level tweaks.

In order to allow web servers to prove their identity, every HTTPS-enabled web browser ships with a hefty set of public keys belonging to a variety of *certificate authorities*. Certificate authorities are organizations that are trusted by browser vendors to cryptographically attest that a particular public key belongs to a particular site, hopefully after validating the identity of the person who requests such attestation and after verifying his claim to the domain in question.

The set of trusted organizations is diverse, arbitrary, and not particularly well documented, which often prompts valid criticisms. But in the end, the system usually does the job reasonably well. Only a handful of bloopers have been documented so far (including a recent high-profile compromise of a company named Comodo[22]), and no cases of widespread abuse of CA privileges are on the record.

As to the actual implementation, when establishing a new HTTPS connection, the browser receives a signed public key from the server, verifies the signature (which can't be forged without having access to the CA's private key), checks that the signed *cn* (common name) or *subjectAltName* fields in the certificate indicate that this certificate is issued for the server the browser wants to talk to, and confirms that the key is not listed on a public revocation list (for example, due to being compromised or obtained fraudulently). If everything checks out, the browser can proceed by encrypting messages to the server with that public key and be certain that only that specific party will be able to decrypt them.

Normally, the client remains anonymous: It generates a temporary encryption key, but that process does not prove the client's identity. Such a proof can be arranged, though. Client certificates are embraced internally by certain organizations and are adopted on a national level in several countries

[*] Public key cryptography relies on asymmetrical encryption algorithms to create a pair of keys: a private one, kept secret by the owner and required to decrypt messages, and a public one, broadcast to the world and useful only to encrypt traffic to that recipient, not to decrypt it.

around the world (e.g., for e-government services). Since the usual purpose of a client certificate is to provide some information about the real-world identity of the user, browsers usually prompt before sending them to newly encountered sites, for privacy reasons; beyond that, the certificate may act as yet another form of ambient authority.

It is worth noting that although HTTPS as such is a sound scheme that resists both passive and active attackers, it does very little to hide the evidence of access to a priori public information. It does not mask the rough HTTP request and response sizes, traffic directions, and timing patterns in a typical browsing session, thus making it possible for unsophisticated, passive attackers to figure out, for example, which embarrassing page on Wikipedia is being viewed by the victim over an encrypted channel. In fact, in one extreme case, Microsoft researchers illustrated the use of such packet profiling to reconstruct user keystrokes in an online application.[23]

Extended Validation Certificates

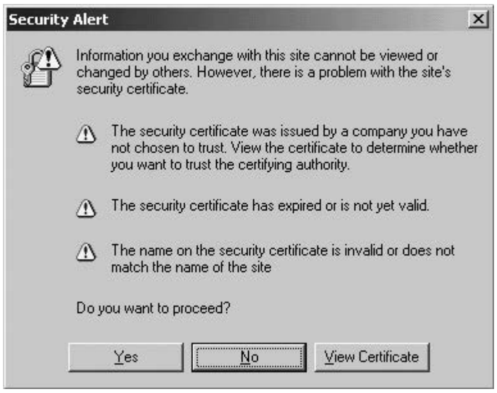
In the early days of HTTPS, many public certificate authorities relied on fairly pedantic and cumbersome user identity and domain ownership checks before they would sign a certificate. Unfortunately, in pursuit of convenience and in the interest of lowering prices, some now require little more than a valid credit card and the ability to put a file on the destination server in order to complete the verification process. This approach renders most of the certificate fields other than *cn* and *subjectAltName* untrustworthy.

To address this problem, a new type of certificate, tagged using a special flag, is being marketed today at a significantly higher price: *Extended Validation SSL (EV SSL)*. These certificates are expected not only to prove domain ownership but also more reliably attest to the identity of the requesting party, following a manual verification process. EV SSL is recognized by all modern browsers by making portion of the address bar blue or green. Although having this tier of certificates is valuable, the idea of coupling a higher-priced certificate with an indicator that vaguely implies a "higher level of security" is often criticized as a cleverly disguised money-making scheme.

Error-Handling Rules

In an ideal world, HTTPS connections that involve a suspicious certificate error, such as a grossly mismatched hostname or an unrecognized certification authority, should simply result in a failure to establish the connection. Less-suspicious errors, such as a recently expired certificate or a hostname mismatch, perhaps could be accompanied by just a gentle warning.

Unfortunately, most browsers have indiscriminately delegated the responsibility for understanding the problem to the user, trying hard (and ultimately failing) to explain cryptography in layman's terms and requiring the user to make a binary decision: Do you actually want to see this page or not? (Figure 3-1 shows one such prompt.)



Security Alert

Information you exchange with this site cannot be viewed or changed by others. However, there is a problem with the site's security certificate.

The security certificate was issued by a company you have not chosen to trust. View the certificate to determine whether you want to trust the certifying authority.

The security certificate has expired or is not yet valid.

The name on the security certificate is invalid or does not match the name of the site

Do you want to proceed?

Figure 3-1: An example certificate warning dialog in the still-popular Internet Explorer 6

The language and appearance of SSL warnings has evolved through the years toward increasingly dumbed-down (but still problematic) explanations of the problem and more complicated actions required to bypass the warning. This trend may be misguided: Studies show that over 50 percent of even the most frightening and disruptive warnings are clicked through.[24] It is easy to blame the users, but ultimately, we may be asking them the wrong questions and offering exactly the wrong choices. Simply, if it is believed that clicking through the warning is advantageous in some cases, offering to open the page in a clearly labeled "sandbox" mode, where the harm is limited, would be a more sensible solution. And if there is no such belief, any override capabilities should be eliminated entirely (a goal sought by *Strict Transport Security*, an experimental mechanism that will be discussed in Chapter 16).

Security Engineering Cheat Sheet

When Handling User-Controlled Filenames in Content-Disposition Headers

☑ **If you do not need non-Latin characters:** Strip or substitute any characters except for alphanumerics, ".", "-", and "_". To protect your users against potentially harmful or deceptive filenames, you may also want to confirm that at least the first character is alphanumeric and substitute all but the rightmost period with something else (e.g., an underscore).

Keep in mind that allowing quotes, semicolons, backslashes, and control characters (0x00–0x1F) will introduce vulnerabilities.

☑ **If you need non-Latin names:** You must use RFC 2047, RFC 2231, or URL-style percent encoding in a browser-dependent manner. Make sure to filter out control characters (0x00–0x1F) and escape any semicolons, backslashes, and quotes.

When Putting User Data in HTTP Cookies

☑ **Percent-encode everything except for alphanumerics.** Better yet, use base64. Stray quote characters, control characters (0x00–0x1F), high-bit characters (0x80–0xFF), commas, semicolons, and backslashes may allow new cookie values to be injected or the meaning and scope of existing cookies to be altered.

When Sending User-Controlled Location Headers

☑ **Consult the cheat sheet in Chapter 2.** Parse and normalize the URL, and confirm that the scheme is on a whitelist of permissible values and that you are comfortable redirecting to the specified host.

Make sure that any control and high-bit characters are escaped properly. Use Punycode for hostnames and percent-encoding for the remainder of the URL.

When Sending User-Controlled Redirect Headers

☑ **Follow the advice provided for Location.** Note that semicolons are unsafe in this header and cannot be escaped reliably, but they also happen to have a special meaning in some URLs. Your choice is to reject such URLs altogether or to percent-encode the ";" character, thereby violating the RFC-mandated syntax rules.

When Constructing Other Types of User-Controlled Requests or Responses

☑ **Examine the syntax and potential side effects of the header in question.** In general, be mindful of control and high-bit characters, commas, quotes, backslashes, and semicolons; other characters or strings may be of concern on a case-by-case basis. Escape or substitute these values as appropriate.

☑ **When building a new HTTP client, server, or proxy:** Do not create a new implementation unless you absolutely have to. If you can't help it, read this chapter thoroughly and aim to mimic an existing mainstream implementation closely. If possible, ignore the RFC-provided advice about fault tolerance and bail out if you encounter any syntax ambiguities.