Rule 2

Use POST requests when actions have side effects

In a GET request, any parameters are encoded as part of the URL. In a POST request, the parameters are "hidden". Where do those parameters go? Let's examine a typical POST request, which may look like this:

```
POST /login.php HTTP/1.0
Host: www.someplace.example
Pragma: no-cache
```

```
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.5a)
Referer: http://www.someplace.example/login.php
Content-type: application/x-www-form-urlencoded
Content-length: 49

username=jdoe&password=BritneySpears&login=Log+in
```

Note the use of `POST` rather than `GET` in the Request-Line. Also, note that this request actually contains data beyond the empty line: 49 bytes, according to the `Content-Length` header. Another header, `Content-Type`, tells the server that these bytes are `application/x-www-form-urlencoded`, as described in RFC 1866 [22].

If you take a closer look at the 49 bytes, you may see that they look exactly like they would look if encoded as part of the URL. And that's what `application/x-www-form-urlencoded` is all about. The parameters are encoded as you are used to, but they are hidden in the request rather than being part of the URL. *URL Encoding* refers to the *escaping* of certain characters by encoding them using a percent sign followed by two hexadecimal digits. Example: We cannot have `AT&T` as part of the query string of a URL, as the ampersand would be taken as a parameter separator. Instead, we URL Encode the troublesome character, and write `AT%26T`, where 26 is the hexadecimal ASCII value of the ampersand.

You have seen the textual nature of a couple of client requests, and a typical server response. Now it's time to talk a little about security. Most of the time, requests are performed by web browsers. But as all requests originate on the client-side, that is, on computers of which the user has full control, nothing stops the attacker from replacing the browser with something completely different. As HTTP borrows its line oriented nature from the telnet protocol [23], you may actually use the `telnet` program to connect to a web server. Try the following command, but replace `www.someplace.example` with something meaningful:

```
telnet www.someplace.example 80
```

Then type in the lines of the first GET request given on page 3 (or paste them in to avoid timeouts). You should get a reply containing, among headers and stuff, the HTML of the root document of the site you connected to.

Instead of using `telnet`, you may write a program to connect a socket and do the actual protocol conversation for you. Anyone capable of writing such

a program has full control over whatever is sent to the web server. And for people who are not able to write such programs themselves, there are freely available programs that will aid them in manipulating all data that get sent to the server [24, 25]. Some of these programs are proxies that sit between your browser and any web server, and that pop up nice dialogs whenever your browser sends anything [26, 27, 28, 29, 30] (see Figure 3.5). The proxies let you change headers and data before they are passed to the server. The server programmer thus can't hide anything on the client-side, and he can't automatically assume that things won't get changed:

Rule 3

In a server-side context, there's no such thing as client-side security