CHAPTER 6

Security and users

6.1 Listening in on the Web

It seems that every few months there are high-profile cases of credit card theft over the Internet; a popular site reports that its customer database was cracked, or a new exploit is discovered that lets a malicious application read information from browsers. As with the case in the physical realm, the bulk of crimes are low-profile and not reported to police. After a pleasant holiday season of shopping over the Web, strange charges turn up on a credit card, and the card holder calls their bank to have the charges removed and to get a new account number issued.

When these cases do make the news, consumers get vague warnings about using proper security when shopping over the Internet. We can hope that those who have been victimized learn their lesson and take precautions when giving out sensitive information.

Seldom, however, is there any comment on the fact that the Internet is not built for security. The most popular protocols for web browsing, email, and file transfer all send their contents without even trivial encryption. The closest physical-world analogy to normal email is to use postcards for all your letters; there isn't a whole lot stopping a snooper from invading your privacy.

Internet protocols send messages in the open primarily because it takes a determined effort to snoop on individual users. For instance, to read a romantic email message from Bob to Carol as it is transmitted, a snooper would need privileged access to Bob's machine, Carol's machine, or one of the machines along the route the message follows. The snooper needs either to listen all the time or to know just when to collect data. If one is really determined to read Bob's love letters, it is probably easier to break into his or Carol's files than to grab the messages on the fly.

On the other hand, if a cracker breaks into a busy Internet service provider (ISP), he can engage in a more opportunistic kind of snooping. By installing a "sniffer" program that reads various kind of Internet traffic, the cracker can look for messages that contain patterns of digits that look like credit card numbers, or phrases like "the password is …" Bob's passion for Carol might escape notice, but he could find his account number stolen the next time he orders something over the Web, only because he or the merchant used the cracked ISP.

Encrypting all Internet traffic sounds tempting at first, but would add expense and delay in the form of additional computation and extra bytes for each message. The most expedient solution is to encrypt traffic which contains sensitive data, and to leave the rest in the open.

This chapter starts with a discussion of Secure Sockets Layer (SSL), the protocol used for most encrypted Internet messages, and how to use it in your web applications. It goes on to cover user authentication schemes and basic user information management issues.

6.2 Secure Sockets Layer (SSL)

Consider the problem of creating secure Internet protocols. One might want to create new protocols for secure HTTP, FTP, or SMTP email, but that would break programs that worked with nonsecure versions.

HTTP and the other protocols are layered on top of TCP/IP, the basic communication layer of the Internet. Most applications that speak TCP/IP do so via *sockets*, which were originally part of BSD Unix but have since been ported to everything from handheld computers to mainframes. Network programmers talk about TCP/IP and sockets almost interchangeably. When a web browser downloads a page from a server, it first opens a socket to the server, which accepts or refuses the connection. Having established communication via TCP/IP, the two then proceed to speak HTTP over the socket connection.[1]

[1] Purists will point out that a socket doesn't have to use TCP/IP, and TCP/IP doesn't have to use sockets. The other common programming interface to TCP/IP is the Transport Layer Interface; interestingly the protocol that is set to supersede SSL is called Transport Layer Security.

By replacing the regular socket library with a secure TCP/IP communication scheme we can leave HTTP alone and still safely transmit sensitive information to and from web browsers. That's the role of the SSL; if the browser and the web server are built with SSL, they can create an encrypted channel and exchange data without fear of snoopers. HTTP rides on top of the layer without additional programming.

SSL is a terrific boon to network applications, but gets surprisingly little use outside of web traffic. Some mail servers and clients support it, but few require it, which is odd, considering that POP and IMAP mail protocols require a username and password to gain access to a server. Those passwords are all being sent in plain text across insecure channels, just as they are for FTP and TELNET sessions (which is why you are using ssh instead). Bob and Carol's true feelings may be known to more people than they realize.

SSL is itself a protocol description with both commercial and Open Source implementations, including SSLeay, a free implementation created by Eric A. Young and Tim J. Hudson, and OpenSSL, which followed on from SSLeay and has become the standard security library for Open Source network products. OpenSSL's developers include members of the Apache Group, so it's no surprise that I'm going to recommend it for use with their server.

6.2.1 Legal issues

You may be aware that there are both patent issues and import/export restrictions on software that uses encryption. In the United States and other countries, commonly used encryption algorithms are patented and require licenses from their patent holders for use. Export restrictions are changing as (some) governments realize that the main effect of legislation is to move encryption development to other countries.

Still, these issues were enough to prevent most US-based sites from distributing encryption software in the 1990s. Distribution web sites generally have guidelines on where to download those libraries, but before doing so you should thoroughly investigate the legalities of their use in your locality.

As the disclaimer goes, I Am Not A Lawyer, but here is my understanding of the legal situation in the United States: the patent holder of the RSA public key encryption algorithm placed the algorithm in the public domain in September 2000 (shortly before the patent was due to expire), so it is no longer necessary to buy a license from RSA or to use the RSAREF implementation. It is legal to use encryption on a US-hosted web site that communicates with the world at large; it may not be legal to let others download your encryption library however.

For hosting in other countries (or browsing, for that matter), see summaries of the legal situation posted at http://www.openssl.org/ for more information although they too will warn you that you need to investigate this on your own.

6.3 OpenSSL and Apache

So now that we know we want OpenSSL, how do we get Apache to use it?

I casually mentioned earlier that a server has to be built to use SSL instead of the usual sockets layer (as do browsers). This is not a trivial change, and can't be implemented solely through an add-on interface to Apache, such as mod_perl is. The guts of the server have to change to handle SSL.

There are commercial Apache SSL products that provide the necessary changes,[2] as well as a pair of Open Source solutions. The first on the scene was Apache-SSL, created by Ben Laurie; later Ralf Engelschall split off the Apache-SSL code to build mod_ssl on an expanded architecture. Both products use OpenSSL, actively track Apache versions (which is not surprising since the developers are part of the Apache Group), use the same license, and accomplish the same goal.

In terms of the buyer's guide, it is hard to tell the two products apart. Their mailing lists are active and helpful. The development pedigree of each product is impeccable and there is no reason to think that one is going to have more ongoing cost than the other. Both products are trivially easy to build and install. The few reports I've read comparing the two implementations comment as much on the developers as the code, so the choice seems to be a matter of personality for those who are active in the development community. I'll put forth a few technical issues and go on with my own choice, mod_ssl. If you choose Apache-SSL instead, the only changes you'll need to make to my examples are in the configuration files.

Both products assume that OpenSSL has been configured and built already. There is some convenience to having all of Apache, OpenSSL, mod_perl, and mod_ssl in one directory tree but it's not a requirement.

6.3.1 Apache-SSL

Apache-SSL provides a set of patches to a given Apache version, plus additional source files. Starting with a freshly unpacked Apache, unpack Apache-SSL into the same directory and apply the patches as instructed. Then configure and build Apache as you have previously, making sure you enable the apache_ssl module as well as mod_perl and any others you use. There isn't much more to it.

There also isn't much more to the documentation. Apache-SSL adds a page to the online manual explaining its directives, and has a configuration example, but doesn't go any further. That's fine for someone who knows about SSL and has a good grasp of Apache configuration, but personally I wanted more.

[2] SSL products from RedHat, Raven, and Stronghold also provided licenses to the patented RSA algorithms for U.S. customers, but that restriction has expired.

6.3.2 mod_ssl

One could argue that the main thing mod_ssl adds to Apache-SSL is polish. The product has an extensive web site which looks better than that of most commercial products. The site has pages for downloading the source, reading the documentation or mailing list archives, getting news about mod_ssl, and checking the latest surveys to track the number of installed servers.

The documentation is quite good, and explains the workings of SSL's suite of cryptographic tools and how a web browser and server decide what to use. The installation instructions that ship with the source are better than the shortened online version, and include instructions on how to build OpenSSL, Apache, mod_ssl, and mod_perl all together. The process isn't that hard to figure out, but having the commands laid out in one file will help the first time web builder.

Those Apache developers who don't like mod_ssl complain that it adds too much to the server. mod_ssl patches Apache to include an extended API, then implements SSL through that API. It also optionally uses the developer's shared memory library to speed up some operations between servers. The result, though, is that mod_ssl acts in many ways like a standard Apache module, and I like the architecture almost as much as I like the generous documentation.

6.3.3 Installing mod_ssl

mod_ssl versions are tied to Apache versions, so if you are downloading newer software, make sure you get the distribution that matches your Apache source.

As mentioned, mod_ssl assumes the current release of OpenSSL is already in place. If you are going to use the MM shared memory library you'll need to set that up as well. This example builds the server using OpenSSL 0.9.5a, Apache 1.3.12, mod_ss 2.6.4, mod_perl 1.24, and MM 1.1.2, all unpacked in `/usr/local`, following the build process as described in the OpenSSL and mod_ssl installation documentation.

```
$ cd /usr/local/openssl-0.9.5a
$ sh config
$ make
$ make test
```

OpenSSL is built with all the defaults, which is fine for the U.S. I moved on to MM, the shared memory module:

```
$ cd ../mm-1.1.2
$ ./configure --disable-shared
$ make
```

The `--disable-shared` directive here disables shared libraries, not shared memory. Since Apache is the only application we're likely to build with MM, there isn't any benefit to having the MM code in a shared library.

Then we'll go to mod_ssl, telling it where to find OpenSSL and MM:

```
$ cd ../mod_ssl-2.6.4-1.3.12
$ ./configure --with-apache=../apache_1.3.12 \
              --with-ssl=../openssl-0.9.5a \
              --with-mm=../mm-1.1.2
```

And on to mod_perl. Here we skip testing mod_perl before going on, but if you've built mod_perl previously that's fine.

```
$ cd ../mod_perl-1.24
$ perl Makefile.PL EVERYTHING=1 APACHE_SRC=../apache_1.3.12/src \
          USE_APACI=1 PREP_HTTPD=1 DO_HTTPD=1
$ make
$ make install
```

Finally, we build Apache. Note the configuration directives for mod_ssl and mod_perl:

```
$ cd ../apache_1.3.12
$ SSL_BASE=../openssl-0.9.5a ./configure --enable-module=ssl \
          --activate-module=src/modules/perl/libperl.a \
          --enable-module=perl
$ make
$ make certificate
$ make install
```

Note the step to create the server's certificate, which we discuss in the next section.

If you have already installed Apache, I recommend shutting down your current server and moving its installation aside, letting `make install` start fresh. Among other things, it will put in a new blank configuration file which has examples of all the SSL directives and an `IfModule` section where you can put SSL-specifics. Compare the newly created `httpd.conf` to your previous one and reinstate your changes (port numbers, aliases, mod_perl configuration, etc.).

6.3.4 Certificates

Before your server can engage in secure sessions it needs a valid certificate which identifies it to browsers. The certificate contains identifying information for your server, a *public key* used to encrypt messages to you, identification of your certifying authority, and a *digital signature* from the certifying authority that the browser can use to prevent malicious servers from forging certificates.

As part of the mod_ssl installation you can create a temporary certificate to be used for testing your server. That will get you through the rest of this book, but before you open your site to the public you will need a proper certificate from a well-known certifying authority (CA).

To explain why this is, we need to delve for a moment into how SSL encrypts Internet messages.[3] When the SSL client contacts an SSL server, the server sends its certificate back along with acknowledgment of the connection. This communication takes

[3] Or at least, how SSL version 2 does so using the RSA algorithm. Version 3 is able to negotiate and use other options for the first secret exchange and the session encryption method.

place in the clear—no secrets have been exchanged yet. Figure 6.1 shows the exchanges in this section.

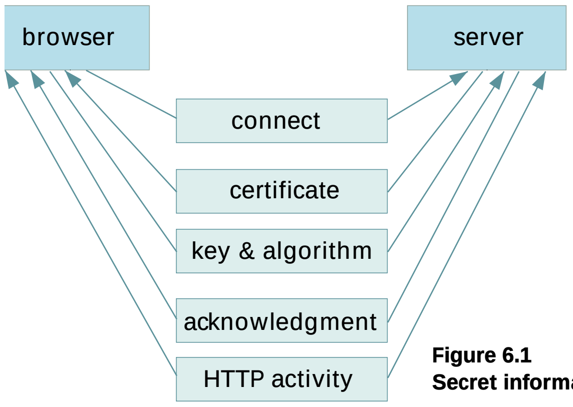

Figure 6.1
Secret information exchange

To establish an encrypted channel, the two parties need to exchange secret information. In particular, they need to decide on a *session key* which will be used to encrypt the data, and the encryption method which applies to it. Obviously if the session key is sent in clear text it won't necessarily stay a secret. To exchange information before establishing the session key, the two parties use *asymmetric* or *public key* encryption.

Public key cryptography, most strongly associated with the RSA algorithm, uses a pair of large numbers called the public and private keys to encrypt and decrypt messages. A message encrypted with the public key can be decrypted with the private key, and a message encrypted with the private key can similarly be decrypted with the public key. The crucial aspect of the algorithm is that a party with only one key can not both encrypt and decrypt a message, thus the term asymmetric encryption. Suppose I give everyone my public key (which is why we call it that) and keep my private key a secret. To send a secure message I encrypt it with my private key, and anyone who has my public key can both decrypt it and also be certain that it came from me, since only I can create a message that can be decrypted with the public key. Anyone who has my public key can send me a secure message also, by encrypting it with that key. Assuming that only I have the secret key, only I will be able to decrypt the result.

Public key cryptography is an amazing example of pure mathematics turned useful. It is also computationally expensive, far too much so to be used for large amounts of web traffic. That's why SSL (and other security protocols) use public key cryptography to exchange an initial message between parties that include a session key for *symmetric* encryption, the more traditional kind of encoding where both parties use a single key to encrypt and decrypt messages. The assumption is that if the session key is used only briefly and is never reused, it is very unlikely that a snooper will figure out the encryption before it is changed again.

Back to our SSL example: the client verifies the certificate (more on this shortly) and uses the server's public key to encrypt a message containing a session key and a choice of algorithm that both parties can use to quickly encrypt and decrypt further messages. The server decrypts the message using its private key, then sends an acknowledgment back that has been encrypted using the session key and a symmetric algorithm.

The role of the certificate, then, is to package up the server's public key along with identifying information used to verify the certificate. This verification is important; it prevents a malicious server from creating a certificate that identifies itself as another party. Part of the certificate is the identity of a CA, which is basically a company that sells trust.

As stated earlier, a certificate contains not only the server's public key and identity but also the CA's identity and digital signature of the public key. That signature is created using the CA's private key to encrypt the server's identification and public key; thus if you have the CA's public key you can decrypt the signature and verify that the server's identification matches their key. Since only the CA's private key can be used to create a valid signature, the browser can trust the certificate—if it trusts the CA, that is.

That sounds like a chicken-and-egg problem, but SSL-enabled web browsers are typically shipped with the identification and public keys of a number of well-known CAs. When a browser gets a certificate signed by an unknown CA it should display a dialog explaining what has happened and warn the user of possible problems; the user can then accept or reject the certificate. Chances are good that you've never seen this warning since sites that use SSL are almost always registered with one of a few popular CAs.

And that leads us to the point of this section: your site must go through the certification process with one of those CAs before you can expect to handle SSL traffic with the Internet at large. Both the Apache-SSL and mod_ssl web sites have lists of CAs which can create certificates for you. The security configuration of your browser will also tell you which CAs your browser will recognize without a warning, so check that list against the list of candidate CAs and then shop around for a good deal with a reputable firm.

As mentioned before, the `make certificate` step will offer to create a temporary certificate for your testing purposes. Go ahead and use that for now; if nothing else it will show you what the warning about an unknown certificate looks like.

6.3.5 Configure and test

If you let `make install` create an `httpd.conf` file, open it and restore your changes to the original for mod_perl and CGI. mod_ssl also will add a new port number section such as:

```
<IfDefine SSL>
   Listen 80
   Listen 443
</IfDefine>
```

The `Listen` directive tells Apache to open additional ports for requests. Port 443 is the standard port for HTTPS (secure HTTP), just as port 80 is the standard for regular HTTP traffic. If your server is listening on some other port for HTTPS, you'll need to specify the port number as part of the URL.

There should also be a new section in the virtual host configuration that looks something like this:

```
<VirtualHost _default_:443>
    #  General setup for the virtual host
    DocumentRoot "/usr/local/apache/htdocs"
    ServerName secure.example.site
    ServerAdmin theo@example.site
    ErrorLog /usr/local/apache/logs/error_log
    TransferLog /usr/local/apache/logs/access_log

    #   Enable/Disable SSL for this virtual host.
    SSLEngine on
</VirtualHost>
```

Apache's `VirtualHost` sections create a sort of server within a server; the parameters to the directive tell Apache when incoming requests are intended for the virtual host. This is typically done by IP address, but can be managed by host name or by port number as shown here. mod_ssl uses a virtual host section to contain directives that apply only to secure HTTP.

A virtual host can have its own document root and log files, and directives placed in this section will apply only to requests for that host. Thus in this case requests that are sent to port 443 will share the usual log files with those sent to port 80, but any error messages will identify the server as secure.example.site. Most importantly, the directive `SSLEngine` turns on SSL communications for port 443.

You can use this section to configure rules that apply only to secure requests. This is a good way to set up applications that require SSL, or to direct users to different applications depending on how they connect. Later we'll use this trick to have one URL display two different pages depending on whether the user makes a secure connection.

After checking and changing your configuration, you are ready to restart Apache. First bring it up in nonsecure mode:

```
/usr/local/apache/bin/apachectl start
```

You should be able to browse the default Apache splash page with your browser. If you have reconfigured your mod_perl and CGI scripts they should work as they did before.

Now shut down Apache and restart it with SSL enabled:

```
/usr/local/apache/bin/apachectl startssl
```

If you encrypted your temporary certificate during the installation, apachectl will prompt you for your pass phrase when you start the server. That's great for security but not practical for a server that needs to be started from a script at boot time. To decrypt your certificate, use the openssl utility that was built as part of OpenSSL:

```
cd /usr/local/apache/conf/ssl.key
cp server.key server.key.crypt
/usr/local/openssl-0.9.5a/apps/openssl rsa -in server.key.crypt -out
server.key
```

Apache will now start without asking for the pass phrase. Make sure that `server.key` is owned by root and that only root can read it.

When Apache starts correctly with SSL enabled you have a secure server. Tell your browser to open https://www.example.site/ to see the default page. Note that URLs beginning with https are directed to port 443 automatically; if you have Apache listening on a different port, you'll need to include the port number in the URL.

The rest of your applications should work fine. Your code can check the `HTTPS` environment variable to determine if it is running in a secure session:

```
if ($ENV{'HTTPS'}) {
    print 'SSL session';
}
else {
    print 'Not secure';
}
```

But we're getting ahead of ourselves. We want a secure channel so we can handle sensitive information, which nearly always means we want to handle user data (as defined in the last chapter). We'll start by identifying the users.

6.4 USER AUTHENTICATION

Novice web programmers are sometimes surprised that web servers have no real idea whom they are talking to when serving up files and running scripts. Programmers who learned their skills on Unix and other user-oriented operating systems are accustomed to having a function that returns a user ID. Mailing lists for most any web development product get questions like "Where do I get the user's name?"

The web server knows the IP address of the requesting browser and an identifying string that should indicate its make and model,[4] but not much more. Most ISPs recycle IP addresses for dial-up users, and even if the address is static there is no guarantee that a particular user will always use the same machine, so this information isn't useful as a user ID.

[4] Applications which use advanced HTML or client-side scripting rely on the `HTTP_USER_AGENT` environment variable to identify the browser, so they can decide on which set of incompatible features to use.

There are two basic approaches to user authentication in a web application: use the authentication protocol built into HTTP or do it yourself.

6.4.1 Using HTTP authentication

Chances are you've already encountered the HTTP authentication protocol already: you request a URL from your browser, and before a new page appears the browser pops up a window or displays a prompt asking for your username and password. That's the authentication protocol in progress.

What's actually going on is more complex than it appears. The protocol works this way:

1. The browser sends the usual request to the web server for a URL.
2. The web server's configuration indicates that authentication is required for that URL. It sends back a 401 response to the browser along with a *realm* for authentication; the realm is a human-readable name used by the server to identify a group of secured documents or applications.
3. If the browser implements authentication caching, it checks its cache for the given realm and server ID. If it already has a username and password for the realm, it uses it to skip the next step.
4. If the browser doesn't have a cache, or the realm isn't there, it displays a dialog box or prompts the user for his username and password for the given realm. The realm should be displayed here so that the user knows which user and password to send.
5. The browser sends the URL request to the server again, including the username and password in the request headers.
6. The server checks the URL, sees that it requires validation (again—remember that this is a stateless protocol), and sees that it has validation headers. It looks up the given username and password in some form of database.
7. If the information is valid, the web server applies the authentication rules for the URL and verifies that the user is authorized to read the associated document or run the application. Everything proceeds as normal if so; if not, it sends back an error page.
8. If the username and password didn't validate, the server sends another 401 response back to the browser, and the cycle continues.

The main advantages of using HTTP authentication is that it already works; Apache has excellent support for it and comes with a few simple user database modules. mod_perl extends Apache with a module that provides authentication against any DBI database, making it trivial to keep your user IDs and other user data together (see the Apache::DBI module's documentation for more information). Many databases (including MySQL and PostgreSQL) have Apache authentication modules as well, so slimmed-down Apache servers can share an authentication database with mod_perl or other applications.

The primary disadvantage of the HTTP authentication mechanism is that it is unfriendly to new users. GUI browsers display a small dialog box prompting for the username and password without much in the way of helpful information. One way to work around this problem is to send a helpful page of information when user authentication fails, instructing the user on how to get an account or what to do at the prompts; this also lets experienced users log in without hand-holding.

HTTP authentication is good for protecting static pages, download directories, or other data for which you would not otherwise write a custom application. It's also fine for administrative functions or private applications when the users will know what to do.

The next section will discuss other reasons to handle authentication yourself. In the meantime, let's look at an example using Apache's simple text file user database.

Suppose we want to protect a page of server status information—the Apache::Status example from the previous chapter. Recall that it was configured in `mod_perl.conf` like so:

```
# Server status
<Location /perl-status>
   SetHandler perl-script
   PerlHandler Apache::Status
   order deny,allow
   deny from all
   allow from 192.168.
</Location>
```

The `deny` and `allow` directives restrict access to a protected network. For purposes of remote administration it would be more helpful to set password protection on the /perl-status URL. The new configuration to handle that is:

```
# Server status
<Location /perl-status>
   SetHandler perl-script
   PerlHandler Apache::Status
   AuthUserFile data/admin_users
   AuthName "Administrator functions"
   AuthType basic
   require valid-user
</Location>
```

Optionally we could keep the `deny` and `allow` directives to further restrict access.

The `AuthUserFile` directive gives the path to a password file to be used in authenticating requests. Remember that all relative file paths begin with Apache's root directory. `AuthName` gives the realm name for authentication, and `AuthType basic` tells Apache that it shouldn't expect the browser to encrypt the information—more on that later in the chapter. The `require valid-user` directive tells Apache that any user with a valid password may retrieve the URL.

Now we need a password file. Apache comes with a utility for creating and managing passwords: htpasswd. Run it with the `-c` switch to create a password file and add a user:

```
/usr/local/apache/bin/htpasswd -c /usr/local/apache/data/admin_users theo
```

The name of the password file matches the path given in `AuthUserFile` earlier (if you add Apache's root directory to the front). The program will prompt for a password, or you can supply one on the command line after the username.

After performing these steps, restart Apache and try the /perl-status URL. If all is well you will be prompted for the user you just created, and then will see the status information page. That's all there is to adding password protection to important pages.

There are more options than shown in this example. For instance, `require` can list a subset of valid users, or specify groups instead of usernames. See the online Apache manual for more information.

The password file created by htpasswd contains user names and encrypted passwords. Make sure that the file is readable by Apache's user. If your applications add users automatically or let them change passwords, then the application's effective user will need write access also.

The text file system is fine for pages that aren't accessed too often and only by a small number of users. To validate the user, Apache has to scan through the file sequentially until it matches the username, so this mechanism will be too slow for a larger user base. Apache comes with hash file authentication modules that are more efficient, but if you have a large user base you probably also have a relational database somewhere. See the examples in the next section for ways to have Apache use your database for authentication.

6.4.2 Doing your own authentication

As you probably know, many web sites have their own login pages, rather than using the mechanism shown in the previous section. Why is that preferable? There are a number of reasons to choose to do your own logins:

- Having your own login page means not relying on browsers to behave correctly. Some browsers had bugs in the caching code for realm information which would cause the browser to send an invalid user and password pair to the server over and over (as the server kept sending back invalid authentication responses). It also allows the application to bypass the cache if the developer wants to force the user to log in.
- I mentioned previously that the standard mechanism is unfriendly to new users. By using your own page, you can control what your users see, provide clear instructions, and tell them what authenticating information you want. You might also use a cookie to store the username and prompt only for the password. I've seen sites that have a custom login page which offers an option to

store both username and password in a cookie, allowing the user to choose less security and more convenience.

- Most browsers indicate when SSL is in use by displaying a lock icon. Letting the user log in from a special page gives a visual reassurance that they are sending their password securely. Alternatively, if you offer the page over both HTTP and HTTPS you can warn users when they are using an insecure connection.
- A login page is one more chance to display announcements, important links, and advertising.

The most obvious disadvantage of writing your own login page is that you have to do the coding yourself, but that's not terribly difficult. The CGI input set includes the password input, which behaves like a regular text input but doesn't echo back the user's characters; and on most OSs Perl includes a `crypt` function which performs quick one-way encryption. (You could store passwords in your database without encrypting them, but then anyone with unrestricted access to the database could steal user passwords, resulting in one of those high-profile cases mentioned at the start of the chapter.)

Here is a very simple login procedure, from `Login.pm`:

```
sub handler {
    # Receive and parse the request.
    my $r = shift;
    my $q = Apache::Request->new($r);
    my $username = $q->param('username') || '';
    my $password = $q->param('password');
```

This is the usual opening code; parse the request, get the expected parameters. The application needs to display a login form, but it needs to display it in more than one case, so the form is built in a variable using Perl's here-doc syntax:

```
    # The login form.
    my $loginForm = <<ENDHTML;
<FORM METHOD="POST">
 <TABLE>
  <TR>
   <TD>Username:</TD>
   <TD><INPUT TYPE="text" NAME="username"
              VALUE="$username" SIZE=10></TD>
  </TR>
  <TR>
   <TD>Password:</TD>
   <TD><INPUT TYPE="password" NAME="password" SIZE=20></TD>
  </TR>
 </TABLE>
 <INPUT TYPE="submit" VALUE="Log in">
</FORM>
ENDHTML
```

The syntax looks odd but it works well once you are used to it. Everything from the line after the label (marked by <<) to the next line beginning with the label is stored in the variable `$loginForm` (or passed to the `print` function in later examples). In this case, the label is `ENDHTML` and the variable contains all those lines of HTML. This syntax works anywhere that a quoted string would work. In fact, it behaves just as double quotes, so that variables in the here-doc are interpolated as they are in a double quoted string. We use that property to put the current value of `$username` into the form. If the application is invoked more than once, the username will be preserved (but the password will not). That's also why `$username` is initialized if the parameter wasn't passed in, to avoid errors about use of uninitialized values.

The here-doc syntax can be used with other forms of quoting; see Perl's perldata documentation for more information. Since web applications often print large blocks of HTML, it is common to see this syntax in Perl CGI or mod_perl scripts. It's not uncommon to use an HTML editor to create the HTML blocks and then cut and paste them into the body of the script. In the next chapter we'll look at other ways to merge Perl and HTML.

Meanwhile, back to the example. Note that the login form uses a password input type (also named password). As you'll see if you run the example, the browser echoes asterisks (or nothing) as the user types there. Also note that the form specifies `METHOD="POST"`. The default method is `GET`, which would pass the username and password as part of the URL (and thus probably display them on the browser's URL line and log them in the server's access log).

Having stored the form, the application can now print it:

```
    # If no username then just display the form.
    unless ($username) {
          my $title = $ENV{'HTTPS'} ?
         'Secure login form' : 'Insecure login form';
        $r->send_http_header('text/html');
        print <<ENDHTML;
<!DOCTYPE HTML PUBLIC>
<HTML>
 <HEAD><TITLE>$title</TITLE></HEAD>
 <BODY>
  <H1>Enter your username and password</H1>
  $loginForm
  To create an account, go <A HREF="/create">here</A>.
 </BODY>
</HTML>
ENDHTML
        return OK;
    }
```

If the username isn't passed in, the application assumes that the user hasn't yet seen the login form. It prints it, then returns `OK` to Apache.

Note that the title of the page depends on whether or not the user is logging in over a secure connection. Of course, a more security-minded application might want to redirect users logging in without SSL to another page, or offer a stronger warning. The title is interpolated into the string via the here-doc syntax again, as is the entire login form. The application also offers a link to the page for creating an account (which we'll get to later in the chapter).

If we did get parameters, we check for the user and password in the database:

```
    # Check the username and password.
    if (defined($username) && defined($password)) {
        my $dbh = DBI->connect('DBI:mysql:Info:localhost',
                               'web','nouser');
        return error($r, "Can't connect to Info") unless $dbh;
        my ($try) = $dbh->selectrow_array
          ('SELECT password FROM Users WHERE username = ?',
             undef, $username);
        if ($try && $try eq crypt($password,$try)) {
          $r->send_http_header('text/html');
          print <<ENDHTML;
<!DOCTYPE HTML PUBLIC>
<HTML>
 <HEAD><TITLE>Hello $username</TITLE></HEAD>
 <BODY>
   Login successful.  Please proceed.
 </BODY>
</HTML>
ENDHTML
        return OK;
}
    }
```

After getting the parameters and checking that we got some kind of value for each, the code connects to the database as it did in the examples from chapter 4. If `DBI->connect` doesn't return a value, the handler calls `error` to display an error page; the code is left out for brevity here.

Assuming we're talking to the database, the code next retrieves the given user's record from the Users table. `$try` is set to the password retrieved from the table. If `$try` is not set, that means there is no such user in the table, and the handler will drop out to the next section.

The handler then calls `crypt` on the password and the value stored in the table. (The need for passing in the already-encrypted value from the database is explained later in this section.) If the given password encrypts to the same value stored in the database, the user is valid and the application can go on to whatever it is going to do.

If not, or if any of the other tests given previously failed, the handler goes on to the next section:

```
    # Invalid user or password.
    $r->send_http_header('text/html');
    print <<ENDHTML;
<!DOCTYPE HTML PUBLIC>
<HTML>
 <HEAD><TITLE>Invalid login</TITLE></HEAD>
 <BODY>
   <H2>The username and/or password you entered is not valid.
   Please try again.</H2>
   $loginForm
  To create an account, go <A HREF="/create">here</A>.
 </BODY>
</HTML>
ENDHTML
    return OK;
}
```

This section simply displays a page indicating that the user can't log in with the given password. It offers the login form again right here, instead of just telling him to go back. The username will be defaulted into the field already but he will have to type his password again. This makes use of the default `ACTION` attribute for the form—the Submit button invokes the same URL again with the current parameters.

The handler needs to be added to mod_perl's configuration:

```
PerlModule Examples::Login
<Location /login>
    SetHandler perl-script
    PerlHandler Examples::Login
</Location>
```

Before trying it, we need to create the Users table. Here is a sample command:

```
CREATE TABLE Users (
       username CHAR(12) NOT NULL,
       password CHAR(13) NOT NULL,
       name CHAR(30),
       email CHAR(24),
       PRIMARY KEY (username)
       );
```

Don't worry about the name and email fields for now. They'll be used by a later example.

We need to add a user to the table, along with an encrypted password. Suppose we want the user to be named 'fred' with password 'tuesday'. We need the encrypted string before we can add Fred.

MySQL has an encryption function, so if that's our database of choice (as shown in the examples) we could use that. But I previously mentioned that Perl makes the `crypt` function available if the operating system provides it, so let's look at a more generic example:

The arguments for `crypt` are the string to encrypt and a two character *salt* value, used by the algorithm in a way similar to the seed of a random number generator.

`crypt` returns a 13 character encrypted string whose first two characters are the salt value. That's why the last block of code retrieved the encrypted password value from the database first before validating the unencrypted test string. The encrypted value was passed along with the test string to `crypt` which looks only at the first two characters of the salt. Without the correct password and salt string it is mathematically unlikely that crypt will generate a matching value.[5]

To create a valid user entry then we need to first create a salt value and encrypt the password with it. We could use any two characters for the salt (as long as they are letters, numbers, or `'.'` or `'/'`), but it would be best to generate a random value for each user. Here is a short Perl procedure from the Examples::CreateUser module that generates the salt and returns a string encrypted with it:

```
sub createPassword {
    my $password = shift;
    my @salts = ('a'..'z', 'A'..'Z', 0..9, '.', '/');
    my $salt = join("", map {$salts[rand(scalar(@salts))]} (1,2));
    return crypt($password, $salt);
}
```

We need to call this function with fred's password. Fortunately, Perl lets us concoct scripts on the command line. By invoking the Examples::CreateUser via the `-M` switch we can call it like so:

```
perl -MExamples::CreateUser -e \
     'print Examples::Createuser::createPassword("tuesday"), "\n"'
PQTpxFNiroTcU
```

Note that you need to either tell Perl where to find Examples::CreateUser via the `-I` switch or run the command from the directory which holds Examples, such as `/usr/local/apache/lib/perl` in my example configuration.

Add user fred with the encrypted password to the database:

```
INSERT INTO Users VALUES ('fred', 'PQTpxFNiroTcU');
```

Now you should be able to log in as fred. Remember that the database user (web) and password given in the example code have to be valid for the database, and the web user must have access to the Users table.

The following examples will make use of `createPassword` in building a user manager.

[5] While it is important to keep the original password a secret, the salt string is less important. There are 4,096 possible salt values (64 choices for each of two characters), and while a human wouldn't want to type them all in, a trivial program could find the right salt given the correct password.

6.4.3 Do I need SSL for this?

Whether you use HTTP authentication or write your own, you face the choice of encrypting the login transaction or not. There are plenty of sites that don't, depending on the situation and what is at risk:

- If the application and its users are protected behind a firewall, encryption is probably not necessary. Conversely, business data which needs to be hidden from employees' casual curiosity should not be on a web site.
- Encryption may be too expensive for high-traffic systems, especially if the user data doesn't contain sensitive information. If you track only user preferences, for example, you (and your users) probably aren't concerned about password theft.
- The HTTP authentication protocol includes another `AuthType`, digest, which does not send passwords in clear text. If your users run browsers that implement digest authentication, then you don't need to add the overhead of SSL. Unfortunately, digest authentication isn't implemented dependably in browsers.

Obviously any e-commerce site which accepts credit cards, Social Security numbers, or other important identifications should use SSL. If your site would be liable to a suit were a user's data stolen, you must encrypt the channel or warn the user of the risk. The simplest practical test is: what could happen to a user whose information is stolen from this site? If your users could be harassed in some way, or your data would help a criminal in committing a crime against them, you are obligated to protect their information.

6.5 USER MANAGEMENT

If a site has user data, it probably needs user management: functions for creating new user accounts and modifying their information.

A site can offer public registration, which allows anyone to create an account and access the site, or have private registration where the potential user fills in a form or sends a mail message to an administrator who then creates an account. Sites that use private registration might still have a web application to create accounts, although the application and other administrative functions would be protected from public use.

To fill out the login handler started earlier, we'll add a page for filling in user information and creating an account. This will also extend the Users table into something more useful and add goodies such as telling the user when the account was last used.

The code samples here are from `CreateUser.pm`:

```
sub handler {
    # Receive and parse the request.
    my $r = shift;
    my $q = Apache::Request->new($r);
    my $username = $q->param('username') || '';
    my $password = $q->param('password');
    my $name = $q->param('name') || '';
    my $email = $q->param('email') || '';
```

This is the standard beginning, checking for and initializing parameters. It also builds the entry form in a variable, using the parameter values:

```
    # The entry form.
    my $form = <<ENDHTML;
<FORM METHOD="POST">
 <TABLE>
  <TR>
   <TD>Desired username:</TD>
   <TD><INPUT TYPE="text" NAME="username"
              VALUE="$username" SIZE=10></TD>
  </TR>
  <TR>
   <TD>Password:</TD>
   <TD><INPUT TYPE="password" NAME="password" SIZE=20></TD>
  </TR>
  <TR>
   <TD>Your real name:</TD>
   <TD><INPUT TYPE="text" NAME="name"
              VALUE="$name" SIZE=30></TD>
  </TR>
  <TR>
   <TD>Your e-mail address:</TD>
   <TD><INPUT TYPE="text" NAME="email"
              VALUE="$email" SIZE=24></TD>
  </TR>
 </TABLE>
 <INPUT TYPE="submit" VALUE="Log in">
</FORM>
ENDHTML
```

The code carries forward the previous values (if any) of the username, name, and email fields, blanking the password (which isn't echoed anyway). You might recall from the examples in chapter 3 that CGI.pm held onto parameter values automatically (it calls them "sticky parameters"). Coding your own HTML means having to take care of this yourself or it means not having to work around CGI.pm features, depending on whom you ask.

If your password creation form doesn't echo password inputs, you should probably require the user to type the password twice and verify that both inputs are the same before creating the account. Some registration systems create a random password for the user and email it to a requested address. In that case, ask for the address twice to help avoid typos.

```
    # Get all the required inputs
    unless ($username && $password) {
        my $title = $ENV{'HTTPS'} ?
          'Secure login form' : 'Insecure login form';
        $r->send_http_header('text/html');
        print <<ENDHTML;
<!DOCTYPE HTML PUBLIC>
<HTML>
 <HEAD><TITLE>$title</TITLE></HEAD>
 <BODY>
  <H2>Please fill in the following information:</H2>
  $form
 </BODY>
</HTML>
ENDHTML
        return OK;
    }
```

This section prints the form if the required fields, username and password in this case, haven't been filled in. A more complex application should give the user a clear indication of what is required.

Note that when we created the Users table we allowed nulls in the name and email fields. We also don't require them to be input here. In general, a field which is required on a form shouldn't allow nulls in a database and vice-versa.

The next section verifies that the username is unique:

```
    # Check for a unique username.
    my $dbh = DBI->connect('DBI:mysql:Info:localhost',
                           'web','nouser');
    return error($r, "Can't connect to Info") unless $dbh;
    my ($try) = $dbh->selectrow_array
      ('SELECT password FROM Users WHERE username = ?',
       undef, $username);
    if ($try) {
        $r->send_http_header('text/html');
        print <<ENDHTML;
<!DOCTYPE HTML PUBLIC>
<HTML>
 <HEAD><TITLE>Already exists</TITLE></HEAD>
 <BODY>
  <H2>User $username already exists.</H2>
  $form
 </BODY>
</HTML>
ENDHTML
        return OK;
    }
```

The retrieval code is the same as that of the previous example, but in this case a successful `SELECT` means that the requested username is taken. It doesn't really matter what we select from the table, so long as we'll know if the record exists. If not, we can go on to add the record:

```
    # Encrypt the password.
    $password = createPassword($password);
    # Write the user record.
    $dbh->do('INSERT INTO Users VALUES (?,?,?,?)',
             undef, $username, $password, $name, $email)
      || return error($r, $dbh->errstr);
```

This uses the `createPassword` function shown earlier to encrypt the password and then store it in the database. DBI's convenient `do` function prepares and executes the given statement. Recall that the first argument after the statement is for attributes, and the rest are placeholder values. There's nothing left to do but confirm things with the user.

```
    # Tell the user that they're one of us now.
    $r->send_http_header('text/html');
    print <<ENDHTML;
<!DOCTYPE HTML PUBLIC>
<HTML>
 <HEAD><TITLE>Account created</TITLE></HEAD>
 <BODY>
  <H2>Account created.</H2>
  Welcome to our site.
 </BODY>
</HTML>
ENDHTML
    return OK;
}
```

Add this handler to your configuration file and restart Apache as usual:

```
PerlModule Examples::CreateUser
<Location /create>
    SetHandler perl-script
    PerlHandler Examples::CreateUser
</Location>
```

Note that the handler is assigned to /create, the URL which was given in the previous login example.

Although the creation example is meant to tie in to the login example, it could also be used by sites with HTTP authentication. The Apache::DBI module includes an authentication hook that validates users in a database using DBI. The creation form could be used to create the necessary record, and regular HTTP authentication takes over from there.

Application designers should bear in mind that users come to a site for content, not user management. If a user requests protected pages without logging in, give them the login page, then go directly to the requested content after the user is validated. HTTP authentication appears to do this automatically (though actually it is the browser's authentication cache that quietly identifies the user for each page); techniques for doing it yourself follow in the next section.

6.6 LOGIN SESSIONS

Users expect web applications to behave as desktop programs: they log in once, and that validation is good until they exit the system. Unfortunately, the nature of web browsing doesn't mesh well with this, particularly in that "exit the system" part.

The typical web approximation is to keep a *login session*: the user is asked for validation the first time he makes a request, and that validation is good as long as the user remains active. If the user doesn't send a request for a certain time period—say, one hour—the session expires and the user is "logged out" (even though in reality he was never logged in).

This kind of validation checking should be unobtrusive, as described at the end of the previous section. We won't require the user to start at a particular page, but will detect when the user needs to log in; having done so, we'll serve up the requested content.

The following example, `Directory.pm`, offers web access to a regular file directory. While this is a very poor way to serve static content, it does demonstrate login sessions and also how a mod_perl application can simulate a file directory (as described in the previous chapter). The program is divided into two functions: `handler` receives the requests and either calls `serveDocument` or displays the login form.

First let's look at `handler`:

```
sub handler {
    # Get the request object and parse it.
    my $r = shift;
    my (%session, @ids, $username);

    # Connect to the database.
    my $dbh = DBI->connect('DBI:mysql:Info:localhost',
                           'web','nouser');
    return error ($r, "Can't connect to database")
      unless $dbh;
```

This is typical opening code. Most paths through the function will use the database, so we open it here for simplicity. Remember that Apache::DBI will be secretly caching database connections, so in most cases this code will simply return a waiting database handle.

Next we check for a session ID:

```
    # Check for a cookie with the ID.
    if (my $cookies = Apache::Cookie->fetch) {
        my $cookieID = $cookies->{'ID'}->value
            if $cookies->{'ID'};
        push @ids, $cookieID if $cookieID;
    }
```

This is the same sort of test we saw in other session examples. As in the previous chapter, we'll check for the session ID both in a cookie and in the URL to support browsers that don't manage cookies for us. The URL is checked for the session ID when we look for the document name:

```
# Examine the URI; it may have a session id,
# and also probably has the requested file.
my $path = $r->uri;
if ($path =~ m{/protected/sessions/(\w+)(/.*)?}) {
    # Session ID is in the URL.
    push @ids, $1;
    $path = $2;
}
elsif ($path =~ m{/protected/(.+)}) {
    # Session ID (if any) will be in a cookie.
    $path = $1;
}
else {
    # Display the directory.
    $path = '';
}
```

If there was a session ID in either, we check that it is a valid session and hasn't expired. Note that this example uses Apache::Session::MySQL to store session info; see the documentation on that module for how to create the required table in your database.

```
# Did we get a session ID somewhere?
if (@ids) {
    my $inSession;
    foreach my $try (@ids) {
        eval {
            tie %session, 'Apache::Session::MySQL', $try,
                {Handle => $dbh, LockHandle => $dbh};
        };
        next if $@;
```

If `eval` noted an error then the session ID isn't valid. Some periodic task should run through the Sessions table and remove data that is too old to be useful. The following shows how this example uses expired sessions:

```
        # Check the expiration.
        if (time() - $session{'time'} < 3600) {
            # Less than one hour, so we're good.
            $inSession++;
            last;
        }
        else {
            # Get the username, then delete it.
            $username = $session{'username'};
            tied(%session)->delete;
            undef(%session);
        }
    }
```

If less than an hour has passed, the session is still good (and as you'll see later, we reset the timer for another hour). If the session is too old, we extract the username from the session so that we can supply it as a default on the login form. Thus we shouldn't delete old sessions from the database too aggressively, since the information is valuable at least in providing this default.

```
        # If we have a session ID, go on to the document.
        if ($inSession) {
            # Display the document.
            return serveDocument($r, $path, \%session);
        }
    }
```

If a valid, unexpired session is waiting, the application calls the function that serves up the requested content. The `serveDocument` function is discussed later in this section.

A more complex application might have a dispatch section here, where the appropriate function is called for the URL or other parameters.

The rest of the handler takes care of the login process:

```
    # If we don't have a session ID (or it expired), the
    # user must log in.  Check for parameters.
    my $q = Apache::Request->new($r);
    $username = $q->param('username') || $username || '';
    my $password = $q->param('password');
    if ($username && $password) {
```

If the handler receives CGI input parameters, then it has already been invoked from the login form; validate those values and proceed.

The initialization of `$username` may look a bit odd here. If it was passed in as a parameter, we want to use that value—that's what the user typed in. If, however, `$q->param('username')` doesn't return a value, then use the value extracted from an expired session (discussed earlier in this section) if there was one, otherwise just set it to an empty string to avoid the "unitialized value" error message when we print `$username`.

The next section validates the username and password as previous examples did:

```
        # Validate the parameters.
        my ($try) = $dbh->selectrow_array
           ('SELECT password FROM Users WHERE username = ?',
             undef, $username);
        if ($try && $try eq crypt($password,$try)) {
            # Everything is valid, so create a session
            # (and set the cookie too).
            eval {
                tie %session, 'Apache::Session::MySQL', undef,
                    {Handle => $dbh, LockHandle => $dbh};
            };

            # Log errors if trying to create a new session failed.
            if ($@) {
                return error($r, $@);
            }
```

This example creates sessions explicitly by passing `undef` to the `tie` call. It reuses the opened database handle (as the section that validated an open session did) for writing the session data to the database.

The final step is to add the username to the session data before displaying the requested content:

```
            # Store the username.
            $session{'username'} = $username;

            # Serve the document.
            return serveDocument($r, $path, \%session);
        }
    }
```

If we get this far, the user hasn't seen the login form yet, or has supplied invalid information. This section displays the form as shown before:

```
    # No username or invalid username/password.
    $r->send_http_header('text/html');
    print '<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">',
          '<HTML><HEAD><TITLE>Please log in</TITLE></HEAD><BODY>';
    if ($password) {
        print '<H2>The password you have entered is invalid.',
              'Please verify your username and continue</H2>.';
    }
    else {
        print '<H2>Please log in.</H2>';
    }

    # Print the form.
    print <<ENDHTML;
<FORM METHOD="POST">
 <TABLE>
  <TR>
   <TD>Username:</TD>
   <TD><INPUT TYPE="text" NAME="username"
              VALUE="$username" SIZE=10></TD>
  </TR>
  <TR>
   <TD>Password:</TD>
   <TD><INPUT TYPE="password" NAME="password" SIZE=20></TD>
  </TR>
 </TABLE>
 <INPUT TYPE="submit" VALUE="Log in">
</FORM>
ENDHTML

    # That's it.
    return OK;
}
```

Note that the handler will use the previous value for the username as a default, but will clear the password each time.

The default action is to request the current URL again when the user clicks the Submit button. We take advantage of that fact here. If the user requests a file but isn't logged in, or his session has expired, he will see this form. After submitting a valid username and password, the maintained URL will be used to look up the requested file.

Note that our session data here is pretty trivial; we could store this much data in a cookie. However, we keep the time stamp data in the session database, where the user can't fake it. Also by expiring sessions quickly we limit the possibility of a snooper stealing an open session. After the session expires, the most a malicious user can learn from an old session ID is the username which created it.

Now let's take a look at the second function, `serveDocument`:

```
sub serveDocument {
    my ($r, $path, $session) = @_;
```

The function receives the request object `($r)`, the document name `($path)`, and the session data as arguments. Note that the caller passed a reference to the session hash: `\%session` in the caller turns into `$session` here.

The function starts by sending a cookie to the browser with a revised expiration:

```
    # Set or refresh the session ID cookie.
    my $cookie = Apache::Cookie->new($r,
                         -name => 'ID',
                         -value => $session->{'_session_id'},
                         -path => '/protected',
                         -expires => '+1h',
                         );
    $cookie->bake;
```

Here the cookie's expiration time is the same as the session expiration time used in the handler function. That makes sense, but it would actually be better to set a longer expiration on the cookie—days or months. There is no sensitive data in the cookie itself; the session ID is carried along to set the default username in the login form.

Alternatively, the application could use two cookies: one for the session ID, set to expire in the right amount of time, and one for the username, with a long lifespan, thus supplying the desired default information.

Depending on the path, the function takes one of two paths:

```
    # Send other headers and start the document.
    $r->send_http_header('text/html');
    print '<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">';
    if ($path) {
        print "<HTML><HEAD><TITLE>$path</TITLE></HEAD><BODY>";
        if (open(DOC, '<', "$docRoot/$path")) {
            print <DOC>;
            close DOC;
        }
        else {
            print "No such document: $path";
        }
    }
```

If the document root directory (stored in a global variable, `$docRoot`, which is initialized outside these functions) and path together form a valid file name, the function opens the file and prints its contents. If not, it displays an error message. That's mostly to catch cases where a user types in a bad URL; the next section shows that the user can get a list of valid files if they don't supply a path:

```
    else {
        # Show the directory.
        print '<HTML><HEAD><TITLE>Directory</TITLE></HEAD>',
              '<BODY><H2>Documents:</H2>';
        opendir DIR, $docRoot;
        while (my $entry = readdir DIR) {
            next unless -f "$docRoot/$entry";
            print '<A HREF="/protected/sessions/',
            $session->{'_session_id'}, "/$entry",
            '">', $entry, '><br>';
        }
        closedir DIR;
    }
```

This section shows the list of documents (all regular files in the document root directory). Note the way the function constructs links to the document, each of the form:

`<A HREF="/protected/sessions/`*`sessionID/filename`*`>`

where *sessionID* is the value also sent in the cookie and *filename* is the actual file name. This supports users whose browsers don't handle cookies, and preserves the file name in the URL. The code in the handler which analyzes the URL will pick up both values and pass them back to this function.

The function refreshes the time stamp in the session and exits:

```
    # Refresh the session timeout.
    $session->{'time'} = time();

    # End the document.
    print '</BODY></HTML>';
    return OK;
}
```

This example didn't require SSL. If the data being served is sensitive data, it would be simple to modify the handler to check the `HTTPS` environment variable and refuse to serve users on unsecured channels. It could also set the security attribute of the session ID cookie to send the ID only when the user connects with SSL.

To run this example, configure the handler in `mod_perl.conf`, then modify the code for an appropriate document root. Don't forget to add the session table to your database if you are going to use Apache::Session::MySQL, which also documents the required table layout. Here is the configuration section:

```
PerlModule Examples::Directory
<Location /protected>
    SetHandler perl-script
    PerlHandler Examples::Directory
</Location>
```

That takes care of security and user administration. The next chapter deals with ways to merge Perl and HTML more cleanly than shown by the long clumsy `print` statements in these examples.