CHAPTER EIGHTEEN

THE PGP, MAIL, AND CGI LIBRARIES

The PGP (**pgp-lib.pl**), mail (**mail-lib.pl**), and CGI (**cgi-lib.pl**) libraries are general libraries that support Web-store-specific functions. For example, since the Web store needs to understand how to process CGI form variables, we use a commonly available library to handle this.

First, the PGP interface library will be discussed in detail. Next, the mail and CGI libraries will be discussed. However, rather than discuss the mail and CGI libraries in depth, this chapter will instead cover the major functions in those libraries that are called. Detailed explanations of the inner workings of these two scripts can be found in our book, *Instant Web Scripts with CGI/Perl* also published by M&T Books (1996).

PGP Interface Library

The purpose of the PGP interface library (**pgp-lib.pl**) is to provide a set of routines that will encrypt orders using the PGP utility. There is just one subroutine in this library, since it only performs one basic duty: encrypting text. The actual use of the PGP Library is discussed in greater detail in Chapter 9.

PGP Library Variables

First, the library defines a set of variables that reflect how PGP is configured on your system. These variables are **$pgp_path**, **$pgp_options**, **$pgp_public_key_user_id**, and **$pgp_config_files**.

- **$pgp_path** is the path to the PGP-executable. This variable includes both the path and filename for the PGP-executable on your system.

```
$pgp_path = "/usr/local/bin/pgp";
```

- **$pgp_options** are the command-line options that will be passed to PGP when it is encrypting the text. The options below configure PGP to accept a file as input (**f**), encrypt the data (**e**), make the encrypted output compatible with ASCII (**a**), store the file in a cross-platform manner with regards to text (**t**), and suppress nearly all informational messages from the program (**+VERBOSE=0**).

```
$pgp_options = "-feat +VERBOSE=0";
```

- **$pgp_public_key_user_id** tells PGP which public key to use for encrypting the text. In the code below, whatever you change **$pgp_public_key_user_id** to becomes your public key for encrypting ordering data in the Web store.

```
$pgp_public_key_user_id = "yourpublickeyid";
```

- **$pgp_config_files** is the path where the PGP configuration files are located. These configuration files include the actual key ring that holds the public key to be used to encrypt ordering information in the Web store.

```
$pgp_config_files = "/home/gunther/public_html/Web_store/
                     Pgpfiles";
```

make_pgp_file Subroutine

The **make_pgp_file** subroutine takes a string (text buffer containing the order) and then returns the encrypted version of that text. It is called by passing it the string along with the path and filename of a directory and filename. This temporary file will hold the encrypted data before it is read back into text variable for returning to the caller of the subroutine.

```
sub make_pgp_file {
  local ($output_text, $output_file) = @_;
  local ($pgp_output);
```

The first thing that must be done in the subroutine is that the **PGPPath** environment variable is set. This environment variable tells the PGP program where to find the configuration files such as the key ring that holds the public key for encrypting data.

```
$ENV{"PGPPATH"} = $pgp_config_files;
```

Next, the actual full PGP command is constructed using the **$pgp_path** variable plus the **$pgp_options**, **$pgp_public_key_user_id**, and a redirection of the output to **$output_file**.

```
$pgp_command =  "$pgp_path $pgp_options ";
$pgp_command .= "$pgp_public_key_user_id ";
$pgp_command .= ">$output_file";
```

The command is opened up as a file using the pipe (|) operator. This tells Perl to actually execute the command but make the command accept input as print statements to the resulting file handle. The output text that must be encrypted is then sent to the command using the **print** statement. Then, the command is closed.

```
open (PGPCOMMAND, "|$pgp_command");
print PGPCOMMAND $output_text;
close (PGPCOMMAND);
```

The resulting output file is then opened and read into the **$pgp_output** variable. The resulting output file basically consists of the encrypted text. When this process is complete, the file is closed and then deleted using the **unlink** command.

```
open(PGPOUTPUT, $output_file);

while (<PGPOUTPUT>) {
   $pgp_output .= $_;
}
close (PGPOUTPUT);

unlink($output_file);
```

Finally, the PGP encrypted text is returned to the caller of the subroutine, and the subroutine as well as the library file is completed.

```
return($pgp_output);
} # End of make_pgp_file
```

E-Mail Library

The email interface library distributed with the Web store actually consists of one of two different files: **smtpmail-lib.pl** or **sendmail-lib.pl**. The UNIX version of Web store uses **sendmail-lib.pl** (renamed **mail-lib.pl** by default) because the most reliable way of sending email on UNIX is through the sendmail program. The Windows NT/Windows 95 version of the Web store uses **smtpmail-lib.pl** instead. This file should be copied over **mail-lib.pl** if you are using a Windows NT or Windows 95 Web server. Both of these files have the same exact *interface* (same function calls), so copying one over the other does not result in a change in the program.

The reason the Windows NT/95 Web servers cannot use the sendmail version of the library is that there is no sendmail equivalent distributed with those operating systems. Thus, instead of interfacing with sendmail, **smtpmail-lib.pl** is programmed to open up "sockets" directly to SMTP (Simple Mail Transfer Protocol) servers on the Internet and send email using the raw, low-level Internet mail protocol.

Unfortunately, using the SMTP version of the library has a downside. The "sendmail" program is a program that does a lot more than just send mail. It has

evolved over the years and has a lot of email error-checking built in to itself. The SMTP version of the mail library on the other hand, while it works, has not seen the same distribution level as the standard UNIX sendmail utility. In addition, using the SMTP version of mail means that the email addresses that you provide to the Web Store configuration must have host names that are the actual Internet names (DNS) of the servers that handle the email directly.

Normally, your Internet address will have a host name after the at symbol (@), which is the actual server that is physically handling the email. However, there are occasions where the host name is actually an alias to another server that is actually handling the mail. The UNIX sendmail program handles this situation automatically. The SMTP library does not. If your email address is **you@yourdomain.com** but the actual hostname of the machine that really handles your email is called **mailserver.yourdomain.com**, then you must put **you@mailserver.yourdomain.com** as your email address in the Web store Setup file.

Other than the difference in handling names, though, the mail libraries are basically identical with regards to the interface to the outside world. There are two main function calls in each library: **send_mail** and **real_send_mail**.

Send_mail Subroutine

The **send_mail** subroutine is the main function that is used by programs when they want to send email. It accepts the "from" address, "to" address, subject, and body of the messages to send, and then sends them. That's all there is to it. The following is an example call to the subroutine assuming that **smith@zzz.org** is emailing to **jones@yyy.org**.

```
&send_mail("smith\@zzz.org", "jones\@yyy.org",
     "Subject". "This is the message\ntosend.\n\n");
```

Real_send_mail Subroutine

The **real_send_mail** subroutine does the same thing as **send_mail** except that it takes more parameters. In fact, **real_send_mail** is called by **send_mail** after **send_mail** breaks down the few parameters it was given in order to fill out the

more complex parameters of **real_send_mail**. The basic difference between the routines is that **real_send_mail** needs to have the host names sent separately from the actual email addresses. This is done so that if the SMTP version of the library is being used, then the programmer will have the option of specifying the real email address in the "from" and "to" areas of the email while still sending email to the correct real hosts that house the actual SMTP server. The equivalent **real_send_mail** call using the data from above appears below in case the **yyy.org** email server is actually physically located at **mail.yyy.org** instead.

```
&real_send_mail("smith\@zzz.org", "zzz.org",
"jones\@yyy.org", "mail.yyy.org", "Subject",
"This is the message\nto send.\n\n");
```

CGI Library

There are certain tasks that every CGI (Common Gateway Interface) program must be capable of doing in order to perform CGI functions. For example, form variables must be read into the program, and specially formatted information must be communicated back to the server. Although there are several good libraries of CGI-related routines, the Web store uses **cgi-lib.pl** because it is small, efficient, Perl 4–compatible, and well supported throughout the CGI and Perl programming community.

cgi-lib.pl was written by Steven Brenner and is the de facto standard library of CGI programming routines for Perl 4 and above. The library is short and simple, and it does 99 percent of what a CGI programmer needs to accomplish with regard to the Common Gateway Interface specification.

The main thing that **cgi-lib.pl** does is to read all the form-variable input into an associative array for picking out the values. It also has the capability of printing standard HTML headers and footers along with the magic **Content-type: text/html\n\n** header. This header is absolutely necessary and printing it is generally the first thing every CGI programmer does in a script. CGI-LIB also has small routines to do general housekeeping, such as printing error messages as HTML output, printing associative arrays, and returning URLs and

certain environmental variables related to CGI. Advanced features, such as the ability to support file uploads, have also recently been added.

The Web store itself only uses a few of the core features of **cgi-lib.pl**. As such, we will only go over those calls that the Web Store actually uses as part of its operation. The operations that the CGI library performs for the Web Store are the processing of form variables and the printing of diagnostic error messages.

Reading and Parsing Form Variables

The **&ReadParse** function is used to read the form variables whether the form was called via a GET or POST method. If **&ReadParse** is called without a parameter, by default the form variables are read into an associative array called **%in**. You can use your own associative array if you pass it by reference to **&ReadParse**. The Web store uses **%form_data** as the associative array name. Thus, **&ReadParse** is called with the following syntax:

```
&ReadParse(*form_data);
```

Normally, when a variable is passed to a subroutine in Perl, only a copy of its value is passed. By replacing the $, @, or % symbol with a * in front of a normal scalar, list array, and associative array variables respectively, you are telling it to copy the location into memory where the variable exists. This way, when the variable is changed in the subroutine, that change is simultaneously done to the originally passed variable instead of to a mere copy of the value of the variable.

Printing CGI Errors

&CgiError and **&CgiDie** accept an error message as a parameter, convert the error message to HTML, and print it so that users can see the error on a Web browser. **&CgiDie** behaves exactly like **&CgiError** except that **&CgiDie** exits the program with the **DIE** command.

Most Perl programs use the plain **DIE** command in case of a system failure. Using **&CgiDie** is much better in the case of CGI programs, because the **DIE** message frequently never gets sent to the browser, making problems difficult to

troubleshoot. The following code gives an example of how you would use **&CgiDie** to trap an error if a file does not open correctly.

```
open(TESTFILE, ">test.file") ||
&CgiDie("The file: test.file could not be opened.");
```

For the code above, you could also have used **CgiError** to report the problem. However, opening a file is usually a crucial step in a program so you want the script to end cleanly instead of attempting to go on when the file could not be opened. Throughout the Web store, **CgiDie** is used instead of **CgiError** for this reason.