CHAPTER SEVENTEEN

THE SETUP CHECKING SCRIPT

The Setup Checking script (**web_store_check_setup.cgi**) was programmed so that a check would be performed on all the Setup file path and file location variables in order to make sure they exist and that their permissions were set correctly. Although you can run this script from the command line of UNIX, the benefit comes from running it through the Web browser as a CGI application. By running the script as a CGI application, the permissions that are checked on the directories and paths are your Web server's permissions rather than your own. This is a true test of your setup and how the main Web store script will react when it is executed by a customer. A discussion of how this script is used and its sample output can be found in Chapter 2.

Global Variable Definition and Setup

The first thing that the CGI script does is tell the Web server to run the script using the Perl executable located in **/usr/local/bin/perl**. If the executable is not located there, however, you will need to change the following line to reflect the real location of where it is located.

```
#!/usr/local/bin/perl
```

The next thing that the script must do is figure out which setup file is being checked. In the case below, **web_store.setup.frames** (The Frames Web store) is the Setup file that will be checked by this script. If you are using a different Setup file, you must change this variable to reflect the new filename.

```
$sc_setup_file = "./Library/web_store.setup.frames";
```

Since this is a script that checks for errors in the Setup, buffering of the output is turned off so that if the script crashes, we make sure that all the output is flushed constantly to the Web browser so that you will know exactly where the problem occurred. In addition, the HTTP header, **"Content-type: text/html\n\n"**, is printed immediately so that all subsequent output will be sent to the Web browser as HTML text.

```
$| = 1;
print "Content-type: text/html\n\n";
```

First, an HTML header is printed to the user's Web browser telling them that this is a script that checks the Web store Setup. The use of **qq!** which is used throughout the Web store indicates that everything between the exclamation points (**!**) is to be printed and that double-quotes do not need to be escaped with a backslash character (\).

```
print qq!
<HTML>
<HEAD>
<TITLE>Check HTML</TITLE>
</HEAD>
<BODY>
<H1>Checking Web Store Setup</H1>
<HR>!;
```

Perl Version and Current Working Directory

For diagnostic purposes, the script checks to see which version of PERL is being used. The special **$]** variable contains the current Perl version. If Perl 4 is being used, the **getcwd.pl** that comes with the standard Perl 4 distribution is required by this script so that it can be used to check what the current working directory is. If Perl 5 is being used, then the **Cwd** package is used instead.

The current working directory is very important for diagnostic purposes. It lets you know where the Web server thinks your script is running, so if the Web store mysteriously cannot find files or require the necessary libraries, you can see

whether this problem is being caused by the Web server thinking that the script is actually in another subdirectory.

```
print "<B>You are using version $] of Perl.<P>";
if ($] >= 5) {
  print "<B>We are about to \"use\" the Cwd package to obtain
    the current working directory</B><P>";
  use Cwd;
  $cwd = getcwd();
} else {
  print "<B>We are about to \"require\" getcwd.pl library to obtain
    the current working directory</B><P>";
  require "getcwd.pl";
  $cwd = getcwd();
}

print "<B>The current working
          directory is: <I>$cwd</I></B><P>";
```

If the Check Setup script dies before displaying the current working directory, then this means that your distribution of Perl is missing some pieces of the standard installation. Your ISP should have installed the current working directory libraries when they installed Perl on your Web server.

Check the Setup File and Load It

Next, the Setup file is checked to see whether it exists and is readable by the Web server. If it is, then it is loaded into memory using the **require** command. If the Setup file does not satisfy this criteria, an error message is printed and the program exits. **$dieflag** indicates whether the error checking should result in the program exiting or not, if an error is encountered in the **Check_path** subroutine.

$dieflag is then set to "off" for dealing with the rest of the file and directory checking through the script. If the Setup file cannot be loaded, this would result in the script exiting anyway, so the **$dieflag** is on for that check. Once the Setup file is loaded, however, the setup check does not exit after every test because if there are multiple problems with the Setup file, the script should inform the user of as many of them that it is able to check.

```
print qq!
<HR>
<H2>Checking: $sc_setup_file</H2>
!;
$dieflag = "on";
&Check_path($sc_setup_file, "sc_setup_file",
"exists,read");
$dieflag = "off";

print "<B>Now, we will load the setup file.</B><P>";
require "$sc_setup_file";
print "<B>Setup File Loaded.</B><HR>";
```

Check the Variables in the Setup File

The following lines in the script check all the variables that were read in from the Setup file using the **Check_path** function. **Check_path** accepts the filename/path variable, a descriptive name of this variable, and a comma-delimited list of permissions that should be checked. The list of possible permissions and attributes to check are **"exists"**, **"read"**, **"execute"**, and **"write"**.

```
&Check_path($sc_cgi_lib_path,
  "sc_cgi_lib_path", "exists,read");
&Check_path($sc_mail_lib_path,
  "sc_mail_lib_path", "exists,read");
&Check_path($sc_html_search_routines_library_path,
  "sc_html_search_routines_library_path", "exists,read");
&Check_path($sc_db_lib_path,
  "sc_db_lib_path", "exists,read");
&Check_path($sc_order_lib_path,
  "sc_order_lib_path","exists,read");
&Check_path($sc_pgp_lib_path,
  "sc_pgp_lib_path", "exists,read");
&Check_path($sc_html_setup_file_path,
  "sc_html_setup_file_path", "exists,read");

&Check_path($sc_user_carts_directory_path,
  "sc_user_carts_directory_path", "exists,read,execute,write");

&Check_path($sc_data_file_path,
  "sc_data_file_path", "exists,read");
$data_path = &get_path_from_full_filename(
```

```
  $sc_data_file_path);
&Check_path($data_path, "path from sc_data_file_path",
  "exists,read,execute");

&Check_path($sc_options_directory_path,
  "sc_options_directory_path", "exists,read,execute");
&Check_path($sc_html_product_directory_path,
  "sc_html_product_directory_path", "exists,read,execute");
&Check_path($sc_html_order_form_path,
  "sc_html_order_form_path", "exists,read");
&Check_path($sc_store_front_path,
  "sc_store_front_path", "exists, read");

&Check_path($sc_counter_file_path,
  "sc_counter_file_path", "exists,read,write");
$data_path = &get_path_from_full_filename(
  $sc_counter_file_path);
&Check_path($data_path, "path from sc_counter_file_path",
  "exists,read,execute,write");

&Check_path($sc_error_log_path,
  "sc_error_log_path", "exists,read,write");
$data_path = &get_path_from_full_filename(
  $sc_error_log_path);
&Check_path($data_path, "path from sc_error_log_path",
  "exists,read,execute,write");

&Check_path($sc_access_log_path,
  "sc_access_log_path", "exists,read,write");
$data_path = &get_path_from_full_filename(
  $sc_access_log_path);
&Check_path($data_path, "path from sc_access_log_path",
  "exists,read,execute,write");
```

The order log path and file information is only checked if the **$sc_send_order_to_log** variable is set to "yes". The same check is done to see if PGP is being used before the PGP-related paths are checked.

```
if ($sc_send_order_to_log =~ /yes/i) {
  &Check_path($sc_order_log_file,
    "sc_order_log_file","exists,read,write");
  $data_path = &get_path_from_full_filename(
    $sc_order_log_file);
```

```
  &Check_path($data_path, "path from sc_order_log_file",
    "exists,read,execute,write");
}

          # If PGP is on, then we will check the
          # pgp related variables
if ($sc_use_pgp =~ /yes/i) {
  &Check_path($sc_pgp_temp_file_path,
    "sc_pgp_temp_file_path", "exists,read,execute,write");
}

&Check_path($sc_root_web_path, "sc_root_web_path",
   "exists,read,execute");
```

Finally, the HTML footer is displayed to the user. This ends the script processing.

```
print qq!
</BODY>
</HTML>!;
```

Check_path Subroutine

The **Check_path** subroutine checks to see if a given path satisfies the attributes/permissions list that was passed to it. If it does not, then the script prints an error. If the criteria is satisfied, then the script tells the user the variable is working properly. **Check_path** takes the actual path (including filename), descriptive variable name (**$varname**), and a comma-delimited list of permissions to check (**$rights**).

```
sub Check_path {
  local($path, $varname, $rights) = @_;
```

If **$rights** has the word **exist** inside its comma-delimited list, the routine checks to see if the file or path actually does exist before checking any other permissions.

```
  if ($rights =~ /exist/i) {
    if (!(-e $path)) {
```

```
  &HTMLDie("<P>Web server thinks $path specified
      in $varname does not exist. <BR>
        <I>This is bad. YOU NEED TO CORRECT
THIS.</I></B><P>\n");
    } else {
  &HTMLMsg("<B>$path specified in $varname exists.<BR>
        <I>This is good.</I></B><P>\n");
    }
  }
```

Next, if the permissions list has the word **read** in it, the file/path is checked to see if it is readable.

```
  if ($rights =~ /read/i) {
    if (!(-r $path)) {
      &HTMLDie("<P>Web server cannot read from $path
      specified in $varname.<BR>
        <I>This is bad. YOU NEED TO CORRECT
THIS.</I></B><P>\n");
    } else {
      &HTMLMsg("<B>The web server can read from $path
                specified in $varname.<BR>
                <I>This is good.</I></B><P>\n");
    }
  }
```

If the permissions list has **write** in it, then the file/path is checked to see if the Web server can write to the directory or file.

```
  if ($rights =~ /write/i) {
    if (!(-w $path)) {
      &HTMLDie("<P>Web server cannot write to $path
      specified in $varname.<BR>
        <I>This is bad. YOU NEED TO CORRECT
THIS.</I></B><P>\n");
    } else {
      &HTMLMsg("<B>The web server can write to $path
                specified in $varname. <BR>
                <I>This is good.</I></B><P>\n");
    }
  }
```

Finally, if **$rights** contains the word **execute**, then the file/path is checked to see if it is executable.

```
  if ($rights =~ /execute/i) {
    if (!(-x $path)) {
      &HTMLDie("<P>Web server cannot execute $path
      specified in $varname.<BR>
        <I>This is bad. YOU NEED TO CORRECT
THIS.</I></B><P>\n");
    } else {
      &HTMLMsg("<B>The web server can execute $path
                specified in $varname.<BR>
                <I>This is good.</I></B><P>\n");
    }
  }
```

The subroutine then prints a horizontal row delimiter (**<HR>**) to the user's Web browser and the subroutine ends.

```
  print "<HR>\n";
} # End of Check_path
```

HTMLMsg Subroutine

HTMLMsg takes a message as an argument and then outputs it in HTML form for this script. This means that the message is changed to have the bold attribute and a paragraph break printed after every message.

```
sub HTMLMsg {
  local($msg) = @_;

  print "<B>$msg</B><P>";
} # End of HTMLMsg
```

HTMLDie Subroutine

HTMLDie does the same thing as **HTMLMsg** except that it also stops the script processing and exits if the **$dieflag** is set to "on".

```
sub HTMLDie {
  local($msg) = @_;
```

```
  print "<B>$msg</B><P>";
  if ($dieflag =~ /on/i) {
    exit;
  }
} # End of HTMLDie
```

get_path_from_full_filename Subroutine

The **get_path_from_full_filename** subroutine takes a full path and filename reference and strips the filename out of it. This is done because anytime there is a file that is being checked in this script, we also generally want to make sure that the directory that it is in has similar permissions. For example, if we check that the error log file is writable, we also want to check that the directory that it is in is writable as well.

```
sub get_path_from_full_filename {
  local($file) = @_;
```

The last **"/"** character is searched in the full filename. If there is one, then the filename is stripped off by truncating the filename variable after the last **/"** in the full filename.

```
  if (rindex($file, "/") > 0) {
    $file = substr($file,0,rindex($file, "/") + 1);
  }

  return($file);
} # end of get_path_from_full_filename
```