# WICKED COOL PERL SCRIPTS

## Useful Perl Scripts That Solve Difficult Problems

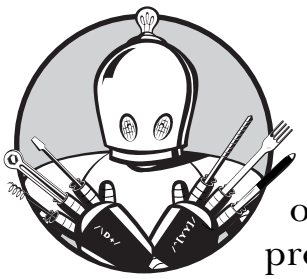
by Steve Oualline

NO STARCH
PRESS

San Francisco

# 3

# CGI DEBUGGING



Perl and the Web were made for each other. The Perl language is ideal for processing text in an environment where speed does not matter. Perl can munch text and use it to produce dynamic web pages with ease.

But programming in a CGI environment is not the easiest thing in the world. There is no built-in CGI debugger. Also, error messages and other information can easily get lost or misplaced. In short, if your program is not perfect, things can get a little weird.

In this chapter, I'll show you some of the Perl hacks you can use to help debug your CGI programs.

## #10 Hello World

This is the CGI version of "Hello World." In spite of it being a very simple program, it is extremely useful. Why? Because if you can run it, you know that your server is properly configured to run CGI programs. And from bitter experience I can tell you that sometimes configuring the server is half the battle.

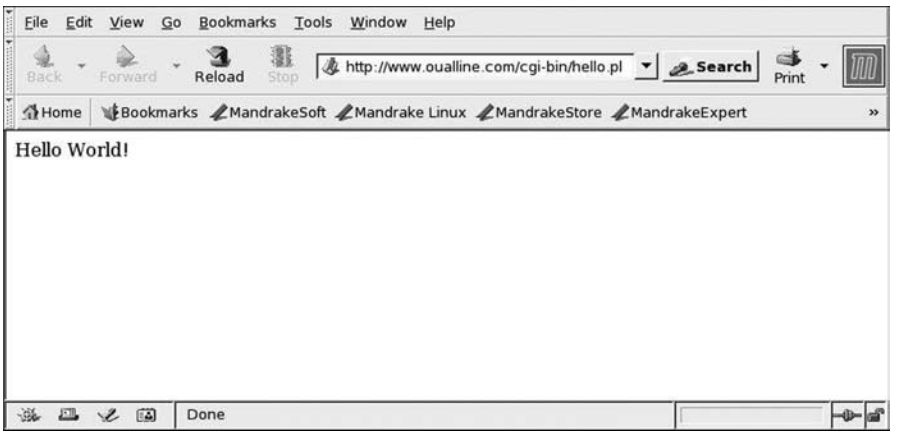
### The Code

```
 1 #!/usr/bin/perl -T
 2
 3 use strict;
 4 use warnings;
 5
 6 print <<EOF
 7 Content-type: text/html
 8
 9 <HEAD><TITLE>Hello</TITLE></HEAD>
10 <BODY>
11 <P>
12 Hello World!
13 </BODY>
14
15 EOF
```

### Running the Script

To run the script, just point your web browser at the correct URL. If you are using the default Apache configuration, the script resides in ~apache/cgi-bin/hello.pl and the URL to run it is http://server/cgi-bin/hello.pl.

### The Results




### How It Works

The script just writes out its greeting, so the script itself is very simple.

The purpose of the script is to help you identify all the problems outside the script that can prevent CGI scripts from running.

### Hacking the Script

In this section, I'm supposed to tell you how to enhance the script. But really, what can you do with "Hello World!"?

I suppose you could enhance it by saying "Hello Solar System," "Hello Galaxy," or "Hello Universe." You are limited only by your imagination.

## #11 Displaying the Error Log

One of the problems with developing CGI scripts is that there's no error displayed when you make a syntax error or other programming mistake. All you get is a screen telling you `Internal Server Error`. That tells you next to nothing.

The real information gets redirected to the error_log file. The messages in this file are extremely useful when it comes to debugging a program.

However, these files are normally only accessible by a few users such as apache and root. These are privileged accounts and you don't want to give everybody access to them.

So we have a problem. Programmers need to see the log files, and the system administrators want to keep the server protected. The solution is to write a short Perl script to let a user view the last few lines of the error_log.

### The Code

```
 1 #!/usr/bin/perl -T
 2 use strict;
 3
 4 use CGI::Thin;
 5 use CGI::Carp  qw(fatalsToBrowser);
 6 use HTML::Entities;
 7
 8 use constant DISPLAY_SIZE => 50;
 9
10
11 # Call the program to print out the stuff
12 print <<EOF ;
13 Content-type: text/html
14 \n
15 <HEAD><TITLE>Error Log</TITLE></HEAD>
16 <BODY BGCOLOR="#FF8080">
17 <H1>Error Log</H1>
18 EOF
19
20 if (not open IN_FILE, "</var/log/httpd/error_log") {
21     print "<P>Could not open error_log\n";
22     exit (0);
23 }
```

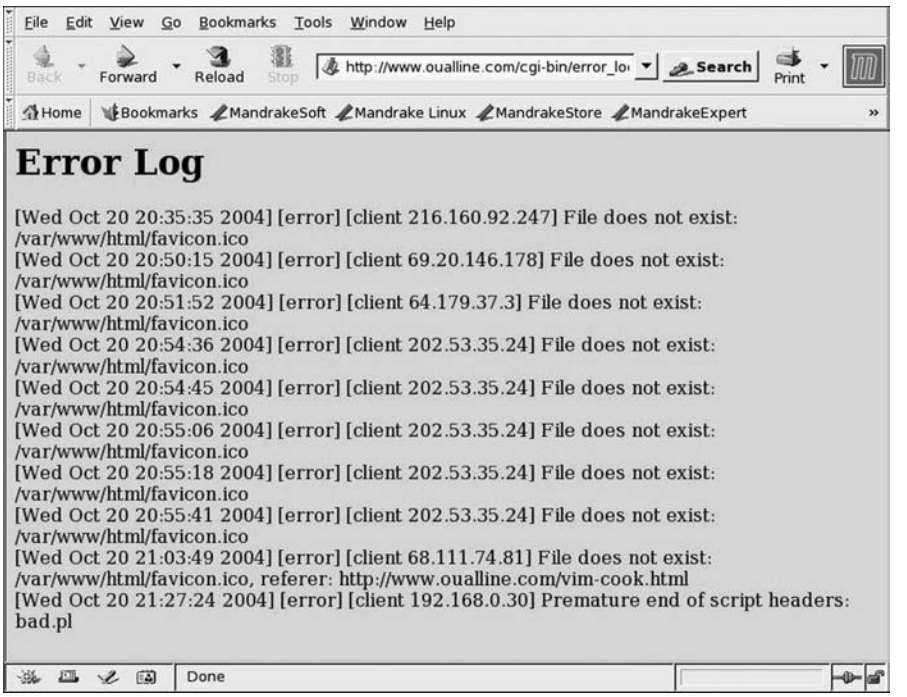
```
24
25
26 # Lines from the file
27 my @lines = <IN_FILE>;
28
29 my $start = $#lines - DISPLAY_SIZE + 1;
30 if ($start < 0) {
31     $start = 0;
32 }
33 for (my $i = $start; $i <= $#lines; ++$i) {
34     print encode_entities($lines[$i]), "<BR>\n";
35 }
```

### Running the Script

The script must be installed in the CGI program directory and must be setuid to root (or some other user who has access to the error logs). It is accessed through a web browser.

### The Results

From this display you can see that the last script run was bad.pl and it errored out because of a `Premature end of script header` error. (Translation: we forgot the `#!/usr/bin/perl` at the top of the script.)



### How It Works

The script starts with the magic line that runs Perl with the `-T` flag. The `-T` tells Perl to turn on *taint* checks. This helps prevent malicious user input from doing something nasty inside your program. It is a good idea to turn on taint for any CGI program. (We'll discuss taint mode in more detail in the next chapter.)

```
1 #!/usr/bin/perl -T
```

The script makes use of the `CGI::Carp` module. This module will catch any fatal errors and print out an error message that is readable by the browser. This means that error messages show up in the browser instead of going only to the error log.

This is especially a good idea for this script. If this script errors out, you can't use the error log script to find out what went wrong (because this is the error log script).

```
5 use CGI::Carp  qw(fatalsToBrowser);
```

Start by outputting a page header. The background color chosen for the errors is #FF8080, which is a sort of sick pink. It looks ugly, but the color screams "Errors!"

```
12 print <<EOF ;
13 Content-type: text/html
14 \n
15 <HEAD><TITLE>Error Log</TITLE></HEAD>
16 <BODY BGCOLOR="#FF8080">
17 <H1>Error Log</H1>
18 EOF
```

Next, open the log file and read all lines in it:

```
26 # Lines from the file
27 my @lines = <IN_FILE>;
```

Finally it's just a matter of printing the last 50 lines. The only trick is that you can't print them directly (they contain text and you want HTML). So the text is processed through the `encode_entities` function to turn nasty ASCII characters into something a browser can understand.

```
33 for (my $i = $start; $i <= $#lines; ++$i) {
34     print encode_entities($lines[$i]), "<BR>\n";
35 }
```

### Hacking the Script

One problem with this script is that it exposes the entire error log to anyone who can access the page. You may want to utilize authentication to prevent unauthorized usage.

Or you can restrict the listing so that only the information for programs created by the user is displayed.

## #12 Printing Debugging Information

CGI programming requires different skills. Not only do you have to know Perl programming, but also HTML and HTML forms. Sometimes what's in the form and what you think is in the form differ. As a result, the inputs to your CGI program aren't what it expects and the program fails.

To help locate errors, it's nice to know the exact inputs to a program. This shows the use of a `debug` function that prints out all the CGI and environment parameters, giving the programmer a lot of extremely useful debugging information.

### The Code

```
#!/usr/bin/perl -T
use strict;

use CGI::Thin;
use CGI::Carp  qw(fatalsToBrowser);
use HTML::Entities;

#
# debug -- print debugging information to the screen
#
sub debug()
{
    print "<H1>DEBUG INFORMATION</H1>\n";
    print "<H2>Form Information</H2>\n";
    my %form_info = Parse_CGI();
    foreach my $cur_key (sort keys %form_info) {
        print "<BR>";
        if (ref $form_info{$cur_key}) {
            foreach my $value (@{$form_info{$cur_key}}) {
                print encode_entities($cur_key), " = ",
                      encode_entities($value), "\n";
            }
        } else {
            print encode_entities($cur_key), " = ",
                  encode_entities(
                      $form_info{$cur_key}), "\n";
```

```
27          }
28      }
29      print "<H2>Environment</H2>\n";
30      foreach my $cur_key (sort keys %ENV) {
31          print "<BR>";
32          print encode_entities($cur_key), " = ",
33          encode_entities($ENV{$cur_key}), "\n";
34      }
35  }
36
37  # Call the program to print out the stuff
38  print "Content-type: text/html\n";
39  print "\n";
40  debug();
```

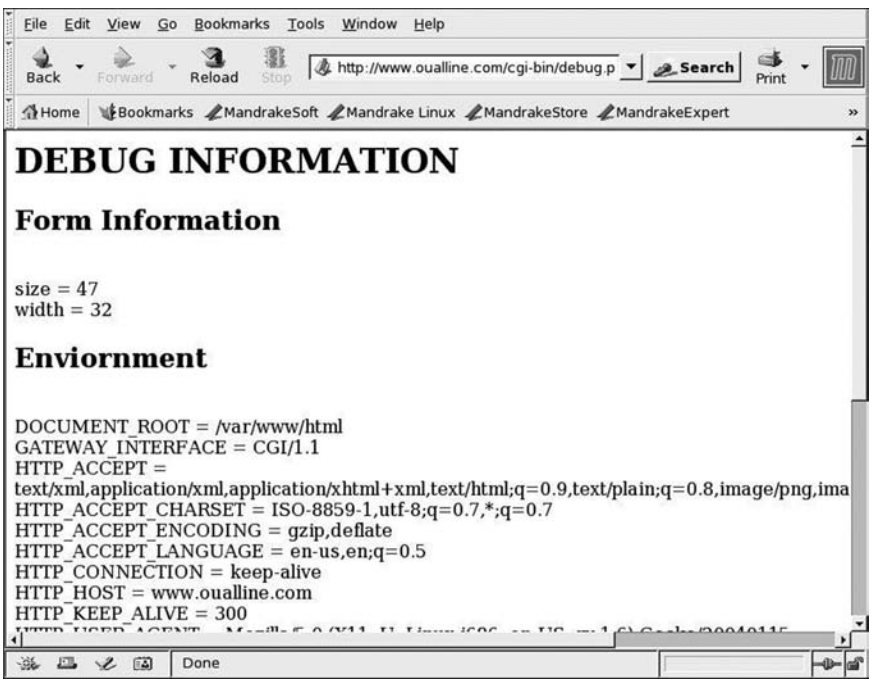
### Using the Function

To use the function, simply put it in your CGI program and call it.

### The Results

Here's the result of running the script. The form we filled in to get to this script took two parameters, a width and a height. From the debug output you can see the values we filled in.

You can also see all the environment information passed to us by the CGI system.



## How It Works

The script uses the `Parse_CGI` function to grab all the CGI parameters. These are stored in the hash `%form_hash`:

```
15      my %form_info = Parse_CGI();
```

The hash creates a

```
form_variable => value
```

mapping. But there is a problem. Some form elements, like a multiple-selection list, can have more than one value. In that case the "value" returned is not a real value but instead a reference to an array of values.

In order to print things, your code needs to know the difference between the two. This is done using the `ref` function. If you have an array reference, you print the elements. If you have something else, you just print the value:

```
16      foreach my $cur_key (sort keys %form_info) {
17          print "<BR>";
18          if (ref $form_info{$cur_key}) {
19              foreach my $value (@{$form_info{$cur_key}}) {
20                  print encode_entities($cur_key), " = ",
21                        encode_entities($value), "\n";
22              }
23          } else {
24              print encode_entities($cur_key), " = ",
25                    encode_entities(
26                        $form_info{$cur_key}), "\n";
27          }
28      }
```

The environment is printed using a similar system. Since you don't have to worry about multiple values this time, the printing is a bit simpler:

```
30      foreach my $cur_key (sort keys %ENV) {
31          print "<BR>";
32          print encode_entities($cur_key), " = ",
33          encode_entities($ENV{$cur_key}), "\n";
34      }
```

Between the environment and the CGI parameters, you've printed every input to a CGI program.

## Hacking the Script

In the field, it would be nice to be able to turn on and off the debugging output at will. One technique is use a remote shell on the server to create a file such as /tmp/cgi_debug and, if it is present, turn on the debugging.

The debug function can also be augmented to print out more information, such as the state of program variables or the contents of information files.

Printing information to the screen is one of the more useful ways of getting debugging information out of a CGI system.

## #13 Debugging a CGI Program Interactively

Perl comes with a good interactive debugger. There's just one problem with it: You have to have a terminal to use it. In the CGI programming environment, there are no terminals.

Fortunately, there is another Perl debug, ptkdb. (The module name is Devel::ptkdb. If you install this module, you've installed the debugger.)

The ptkdb debugger requires a windowing system to run. In other words, if the web server can contact your X server, you can do interactive debugging of your CGI script.

The only trick is how to get things started. That's where this debugging script comes in.

### The Code

```
 1 #!/usr/bin/perl -T
 2 #
 3 # Allows you to debug a script by starting the
 4 # interactive GUI debugger on your X screen.
 5 #
 6 use strict;
 7 use warnings;
 8
 9 $ENV{DISPLAY} = ":0.0"; # Set the name of the display
10 $ENV{PATH}="/bin:/usr/bin:/usr/X11R6/bin:";
11
12 system("/usr/bin/perl -T -d:ptkdb hello.pl");
```

### Running the Script

The first thing you need to do is edit the script and make sure that it sets the environment variable *DISPLAY* to the correct value. The name of the main screen of an X Window System is *host*:0.0, where *host* is the name of the host running the X server. If no host is specified, then the local host is assumed.

**NOTE** *If you are running an X Window System with multiple displays, the display name may be different. But if you're smart enough to connect multiple monitors to your computer, you're smart enough to set the display without help.*

The other thing you'll need to do is to change the name of the program being debugged. In this example, it's hello.pl, but you should use the name of your CGI program.

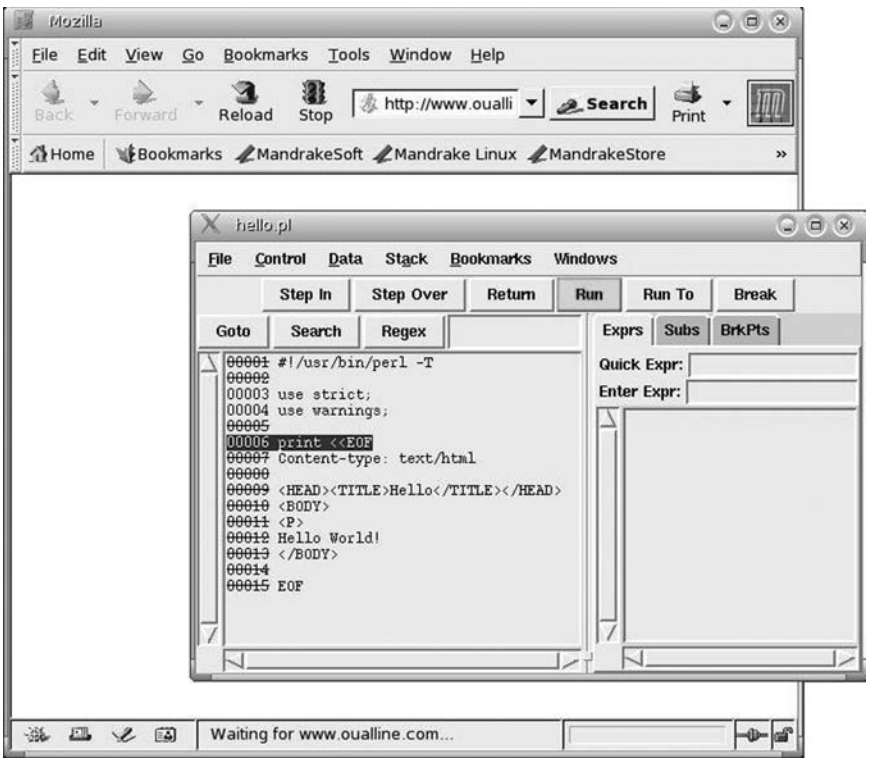
Once you've made these edits and copied the start-debug.pl script into the CGI directory, point your browser at the start-debug.pl script:

```
$ mozilla http://localhost/cgi-bin/start-debug.pl
```

## The Results

The script will start a debugging session on the script you specified.

You can now use the debugger to go through your code step by step in order to find problems.




## How It Works

The simple answer is that it executes the following command:

```
$ perl -d:ptkdb script
```

Unfortunately, there are a few details you have to worry about. First, the script is run with the taint option:

```
1 #!/usr/bin/perl -T
```

Taint mode turns on extra security checks which prevent a Perl program from using user-supplied data in an insecure manner.

Next you set the display so that the debugger knows where to display its window:

```
9 $ENV{DISPLAY} = ":0.0"; # Set the name of the display
```

Because taint checks are turned on, the `system` function will not work. That's because the `system` function uses the `PATH` environment variable to find commands. Since `PATH` comes from the outside, it's tainted and cannot be used for anything critical.

The solution is to reset the path in the script. Once this is done, `PATH` is untainted and the `system` function works:

```
10 $ENV{PATH}="/bin:/usr/bin:/usr/X11R6/bin:";
```

All that's left is to run the real script with debugging enabled:

```
12 system("/usr/bin/perl -T -d:ptkdb hello.pl");
```

### Hacking the Script

This script is extremely limited. It can only debug programs named hello.pl. With a little work, you could create a CGI interface to the front end and make the script debug anything.

This brings us to the other problem with this script: no security. If you can get to the program, you can get to the debugger. From the debugger, you can do a lot of damage. It would be nice if the script let only good people run it.

But as a debugging tool, it's a whole lot better than the usual CGI debugging techniques of hope, pray, and print.