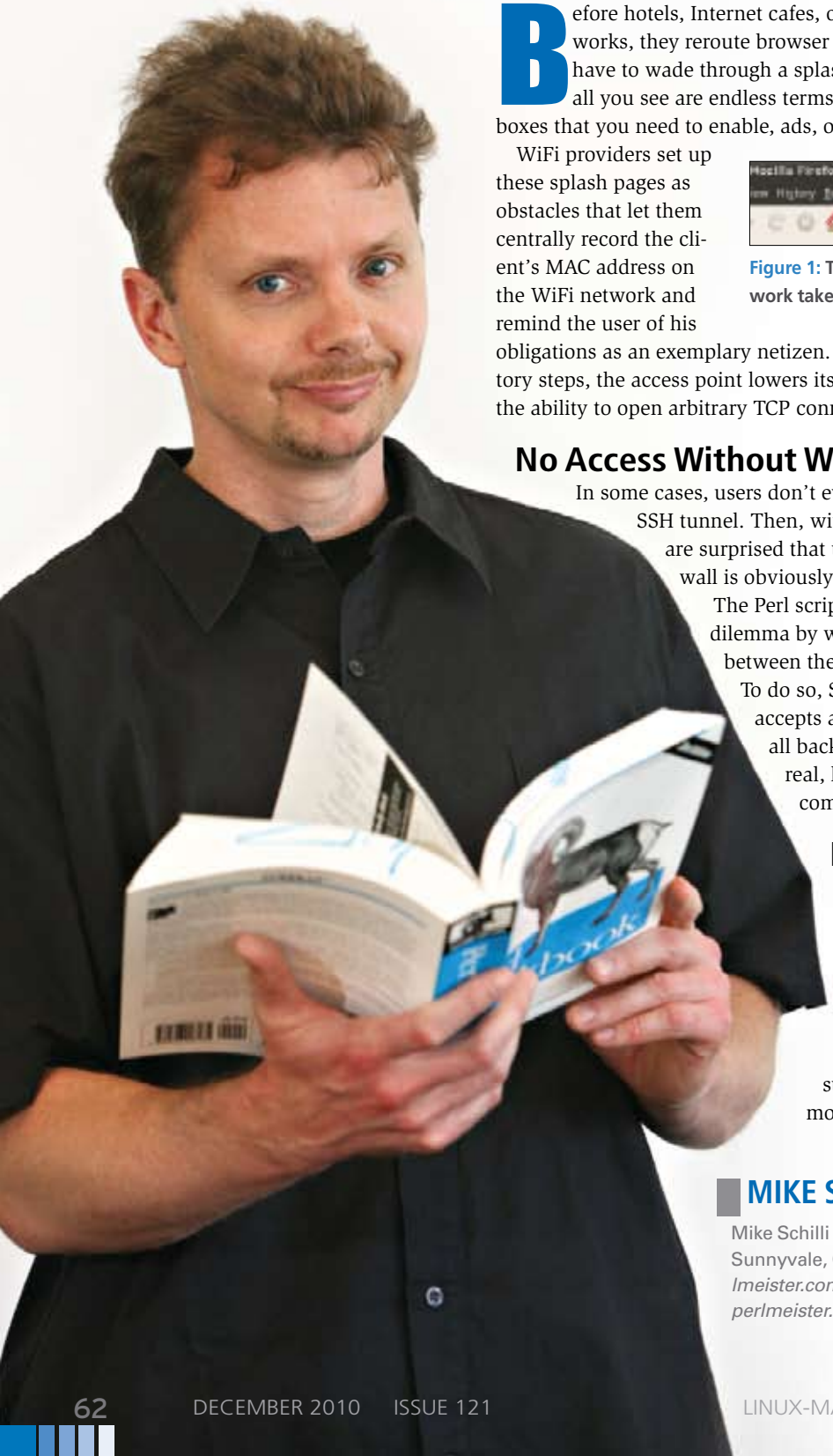
Judith: Please change to: @SW:Perl: Scripting -rls

Judith: I pulled the box with the @D/@T lines up so the 'p' descenders didn't encroach on the @V/@A box. -rls

Bypass annoying WiFi splash pages

Quick Approval

A Perl script saves busy users the bother of clicking through WiFi provider splash pages, automatically accepts the terms of use, and enables access as quickly as possible without a browser. *By Michael Schilli*



Before hotels, Internet cafes, or airports let new users access their WiFi networks, they reroute browser requests (Figure 1), and potential surfers first have to wade through a splash page (Figure 2). When you get to that page, all you see are endless terms of use, which nobody reads anyway, checkboxes that you need to enable, ads, or all three.

WiFi providers set up these splash pages as obstacles that let them centrally record the client's MAC address on the WiFi network and remind the user of his obligations as an exemplary netizen. Once you have been through all these mandatory steps, the access point lowers its drawbridge and gives you both web access and the ability to open arbitrary TCP connections to the Internet (Figure 3).

Figure 1: The user requested google.com, but first the WiFi network takes him to the provider's splash page.

No Access Without Web

In some cases, users don't even want web access; they just want to open an SSH tunnel. Then, with the shell open and the browser closed, users are surprised that they are assigned an IP address although the firewall is obviously blocking access to the Internet.

The Perl script `splash` described here helps users out of this dilemma by worming its way through the web forms that lie between the raring-to-go user and full access to the Internet. To do so, Splash clicks all the links, checks all the boxes, accepts all the cookies, and well-behavedly sends them all back to the WiFi provider, who then thinks that a real, live human is using their browser for all of this communication.

Flexible Plugin Tactics

But how can a simple script handle what could be thousands of different splash page configurations? WiFi providers might create complex forms or even use their heavy artillery like JavaScript or Flash animations to fend off automatic scripts.

To counter this, the script relies on a plugin strategy that allows the user, or a third party, to modify it to handle new methods. Each plugin tries

MIKE SCHILLI

Mike Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He can be contacted at *mschilli@perlmeister.com*. Mike's homepage can be found at *http://perlmeister.com*.

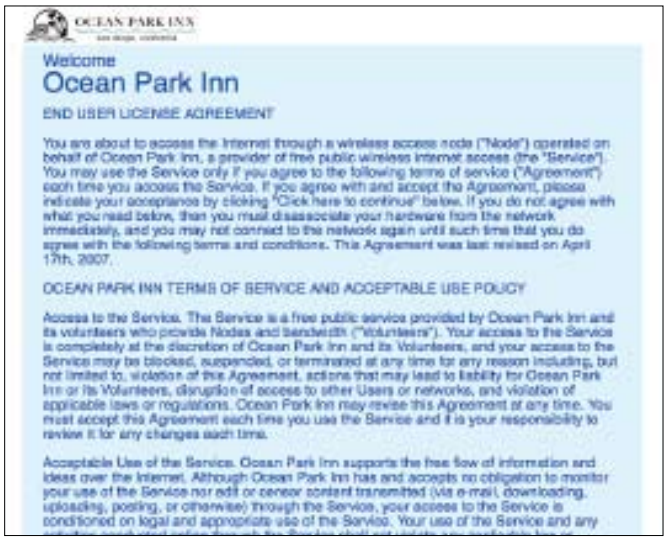


Figure 2: The hotel WiFi rerouting the first browser request to a splash page.

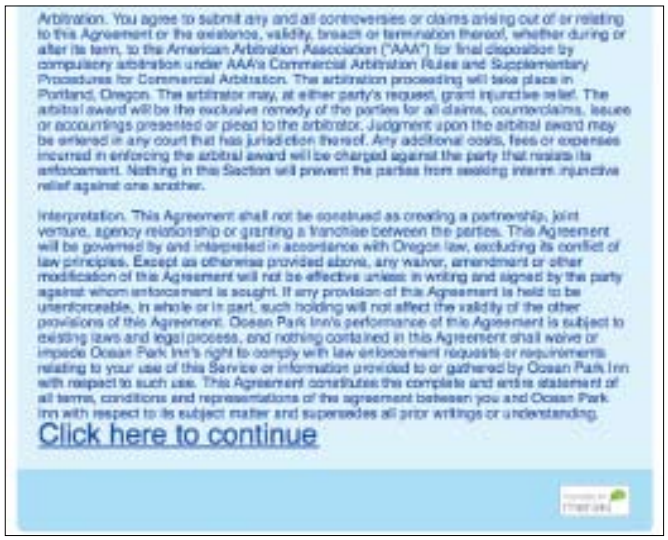


Figure 3: Users just need to click to access the terms of use, and can then surf for 24 hours.

its own approach to outflanking the splash page.

For example, the ClickAllLinks.pm plugin clicks its way through all the links on the page, whereas CheckBoxFill.pm enables all the checkboxes displayed by the first web form before pressing the submit button. Users simply add newly developed plugins to the plugin directory, then the script automatically picks them up, without any need for configuration, and tries them out as a new tactic.

Whenever one plugin completes, the Splash script checks to see whether a request for *http://www.google.com* really returns the page for the website or whether another splash page is in the way. If the access attempt is unsuccessful, the script tries the next plugin; otherwise, it outputs a success message and quits.

Down with the Wall

Figure 4 shows how the script reacts if the network connection is still down because the user hasn't selected any of the available WiFi networks or entered the WPA password. While the Google request returns errors, or freezes for more than five seconds, the script takes a short break and then tries again. Once the client has an IP address and web requests at least show the splash page, the script throws one plugin after another at the artificial wall (Figure 5) until it finally comes tumbling down.

Listing 1 shows the `SplashJumper` base class for all the plugins, which simply defines a constructor. It pulls in the CPAN Module::Pluggable module and passes the `require => 1` parameter setting to it. The module then checks the `SplashJumper/Plugin` subdirectory for `.pm` files, issues a `require` instruction, and loads them into the active script.

The Best First

Listing 2 shows a typical plugin with the ClickAllLinks module. When the main

LISTING 1: SplashJumper.pm

```
01 ##############################
02 package SplashJumper;
03 ##############################
04 # Mike Schilli, 2010
05 # (m@perlmeister.com)
06 use strict;
07 use warnings;
08 use Module::Pluggable
09   require => 1;
10
11 sub new {
12  bless {}, shift;
13 }
14
15 1;
```

LISTING 2: ClickAllLinks.pm

```
01 ##############################
02 package SplashJumper::Plugin::
   ClickAllLinks;
03 ##############################
04 # Mike Schilli, 2010
05 # (m@perlmeister.com)
06 ##############################
07 use Log::Log4perl qw(:easy);
08
09 ##############################
10 sub register {
11 ##############################
12  return "click-all-links",
13    10;
14 }
15
16 ##############################
17 sub process {
18 ##############################
19  my ($self, $mech) = @_;
20
21  for
22    my $link ($mech->links())
23  {
24
25   INFO "Clicking on ",
26     $link->url();
27   my $resp =
28     $mech->get($link);
29
30   INFO "Got ",
31     length($resp->content()),
32     " bytes back";
33
34   $mech->back();
35  }
36 }
37
38 1;
```

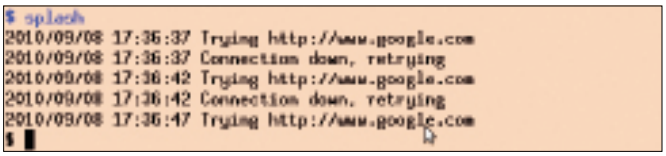


Figure 4: If the network connection is down, the script tries to reach the Google server at intervals of 5 seconds.

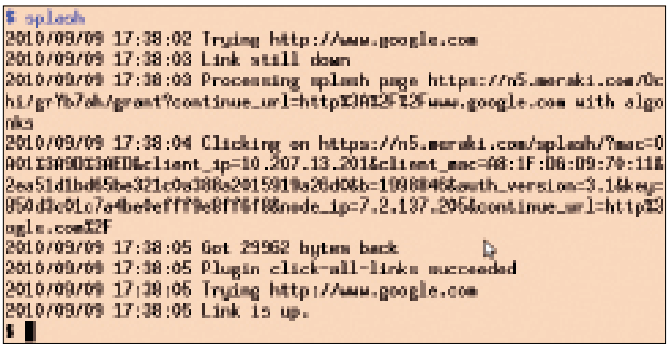


Figure 5: If the script encounters a splash page, it first applies the click-all-links method.

script loads the plugin, it calls the plugin's `register()` method, which returns the name of the current tactic and a priority number. The main script sorts the plugins numerically on the basis of the priority numbers they report, calling plugins with a lower value first. This means that plugins can decide among themselves who goes first. It makes sense to run the plugins with the best chance of success first to reduce the run time.

The two plugins I will look at here define priorities of 10 and 50, respectively, so that the script will always try ClickAllLinks first before applying the CheckBoxFill tactic shown in Listing 3.

When a splash page counterattack plugin runs, the script calls its `process()` method and passes in the `$mech` browser simulator, a WWW::Mechanize class object. This CPAN module is perfect for grabbing web pages, analyzing their content, and drilling down on its links. It is often used to implement screen scrapers, because it provides more or less a full set of browser capabilities (except JavaScript code or Flash plugins), automatically accepts cookies, and returns them to the server.

In the ClickAllLinks plugin, the `links()` method identifies all the links on the retrieved splash page in the form of WWW::Mechanize::Link objects and uses a `for` loop to iterate over the list. The `INFO` function, imported from the Log4perl treasure trove, tells the inquisitive user which link the plugin is currently clicking. The browser simulator's `get()` method then executes the web request.

Mechanize with Autocheck

Explicit error checking is not performed here because WWW::Mechanize runs in autocheck mode by default; it throws an exception for any errors it encounters, and the `eval{}` block wrapped around the call to the plugin in the main script (Listing 4, line 55) fields them before the main program goes on to handle the error.

LISTING 3: CheckBoxFill.pm

```
01 ##############################
02 package SplashJumper::Plugin::
   CheckBoxFill;
03 ##############################
04 # Mike Schilli, 2010
05 # (m@perlmeister.com)
06 ##############################
07 use Log::Log4perl qw(:easy);
08
09 ##############################
10 sub register {
11 ##############################
12  return "checkbox-fill", 50;
13 }
14
15 ##############################
16 sub process {
17 ##############################
18  my ($self, $mech) = @_;
19
20  $mech->form_number(1);
21
22  my @inputs =
23    $mech->current_form
24    ->find_input(undef,
25   "checkbox");
26
27  for my $input (@inputs) {
28   $input->check();
29  }
30
31  INFO "Submitting form 1";
32  $mech->submit_form(
33   form_number => 1);
34  $mech->back();
35 }
36
37 1;
```

In line 30 (Listing 2), the plugin then uses another `INFO` instruction to indicate the number of bytes returned by clicking the link, and the `back()` method in line 34 presses the virtual browser's *Back* button to return to the splash page.

The CheckBoxFill plugin in Listing 3 uses a different approach. It searches through the HTML on the splash page to find the first web form and then uses the `form_number(1)` instruction to set it as the `current_form`. The `find_input()` method extracts all the `"checkbox"` type input fields from the form, and the `for` loop in lines 27-29 selects all of them by calling their `check()` methods. Line 32 then calls `submit_form()` to send the web form back to the server, before line 34 returns to the splash page to make sure that the next plugin finds normal starting conditions.

This method makes it possible to handle splash pages like those shown in Figures 6 and 7, both of which require the user to check a box and then click a button that submits the form.

Listing 4 finally shows the main program, Splash. It starts by defining in line 13 the test URL, which the script retrieves to discover whether the Internet connection is up. The Google page is perfect for this, because it is lightweight and highly likely to be available on a working Internet. Line 16 initializes the Log4perl framework at level `DEBUG` to give the user detailed information on the plugin that is currently running and the actions that it is performing in the virtual browser.

Thanks to using Module::Pluggable, `SplashJumper` base class objects feature a `plugins()` method that returns a list of all the plugins installed in the `Plugin` subdirectory. Line 28 checks whether each plugin provides a `register()`

method, as required, and omits any plugins that are not correctly implemented, returning an error message to this effect at the same time.

Correctly implemented plugins return their tactics as `$algo` in line 35 and their preferred numeric priority in the `$order` variable. The script bundles the data it finds into an array and pushes a reference to it to the end of the `@ways` array. Line 44 sorts the elements in this array numerically by the `order` field so that a plugin with a priority of 10 runs *before* a plugin with a priority of 50.

The WWW::Mechanize browser simulator sets its timeout to five seconds in line 50; this means that the loop block in lines 53-62 only freezes in the `get()` method for five seconds before giving up, sleeping for five seconds, and then trying again to reach the Google server. At the end of the block, which Perl repeats by issuing a `redo`, at least the local WiFi network is working and the client was assigned a valid IP address; however, the WiFi provider still might have routed requests to *www.google.com* to an internal server that produced the splash page.

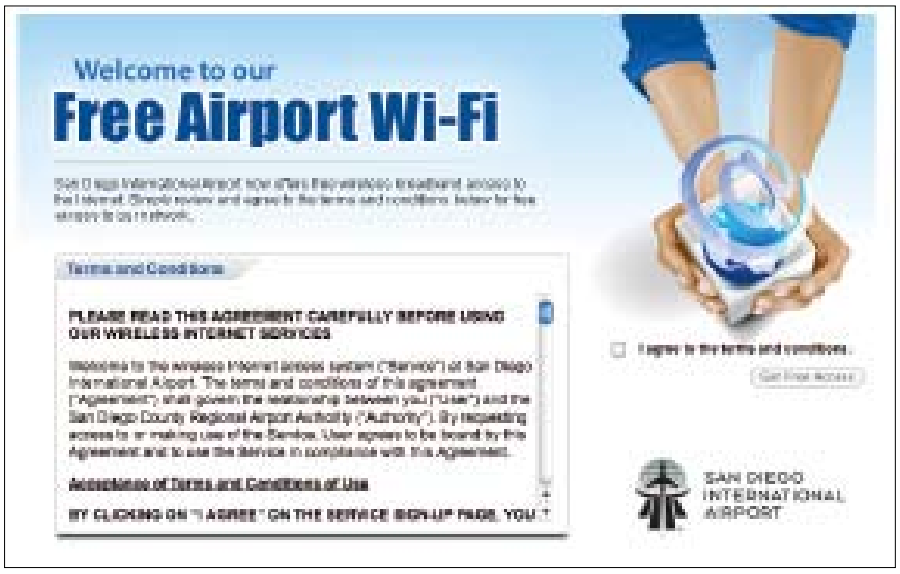


Figure 6: The free WiFi network at San Diego airport requires users to check a box and then submit the web form.

The `for` loop in lines 65-95 attempts to outwit the splash page with a variety of plugins, and once the URL for the latest request is equal to the test URL (i.e., no redirect to the splash page happened), line 72 decides that the splash page has been defeated and the Internet connection is open.

If this is not the case, line 85 calls the `process()` method for the next plugin in

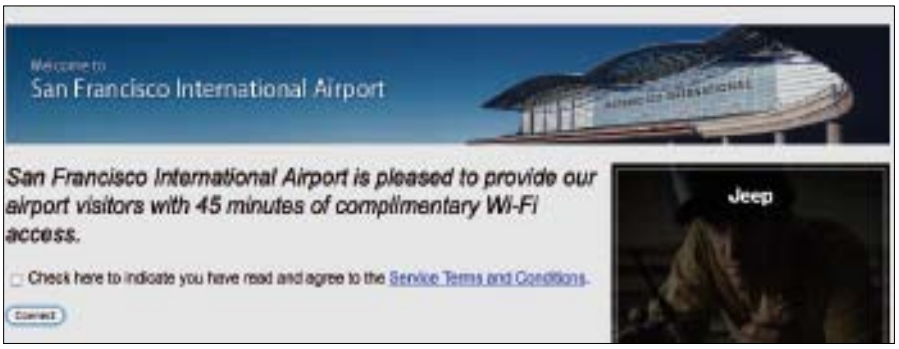


Figure 7: The free WiFi at San Francisco airport also features a checkbox.

line with equal or higher priority. The `eval{}` block in lines 81-86 traps any errors that occur in the plugin, and line 88 checks the `$@` variable to see if anything has happened.

Installation

To makes sure the script finds the SplashJumper module, the latter must be installed in the script's `%INC` search path: The easiest way to do this is to put both in the same directory. Plugins are stored in the newly created `SplashJumper/Plugin` subdirectory so that the file layout looks like this:

```
splash
SplashJumper.pm
SplashJumper/Plugin/ClickAllLinks.pm
SplashJumper/Plugin/CheckBoxFill.pm
```

Additional plugins could potentially handle multiple web forms, or even work their way around JavaScript tricks. They also need to be stored in the `Plugin` directory. Their `register()` method needs to assign a name for the tactic, and they decide on a priority to define the order in which they will run. After opening your laptop's lid at the airport, you then just need to launch the `splash` script. Its output tells you how the pocket warrior is fairing in its new environment and whether it successfully pushed aside those artificial roadblocks. ■■■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Resources/Article-Code*

LISTING 4: Splash

```
01 #!/usr/local/bin/perl -w
02 ############################
03 # splash - Traverse WiFi
04 #          Splash Pages
05 # Mike Schilli, 2010
06 # (m@perlmeister.com)
07 ############################
08 use strict;
09 use SplashJumper;
10 use WWW::Mechanize;
11 use Log::Log4perl qw(:easy);
12
13 my $url =
14   "http://www.google.com";
15
16 Log::Log4perl->easy_init(
17  $DEBUG);
18
19 my $sj = SplashJumper->new();
20
21 my @ways = ();
22
23 for
24   my $plugin ($sj->plugins())
25 {
26
27  if (
28   !$plugin->can("register"))
29  {
30   ERROR "$plugin can't do",
31         " register()";
32   next;
33  }
34
35  my ($algo, $order) =
36    $plugin->register();
37
38  push @ways,
39    [ $algo, $plugin,
40   $order ];
41 }
42
43 # sort by plugin priority
44 @ways = sort {
45  $a->[2] <=> $b->[2]
46 } @ways;
47
48 my $mech =
49   WWW::Mechanize->new();
50 $mech->timeout(5);
51
52 # wait until network is up
53 {
54  INFO "Trying $url";
55  eval { $mech->get($url); };
56  if ($@) {
57   INFO
58 "Connection down, retrying";
59   sleep 5;
60   redo;
61  }
62 }
63
64 # try to get past splash page
65 for my $ways (@ways) {
66
67  my $current_url =
68    $mech->response->request
69    ->uri;
70
71  if ($current_url eq $url) {
72   INFO "Link is up.";
73   last;
74  } else {
75   INFO "Link still down";
76  }
77
78  my ($algo, $plugin, $order)
79    = @$ways;
80
81  eval {
82   INFO "Processing splash ",
83        "page $current_url ",
84        "with algo $algo";
85   $plugin->process($mech);
86  };
87
88  if ($@) {
89   ERROR
90     "Algo $algo failed ($@)";
91  } else {
92   INFO
93     "Plugin $algo succeeded";
94  }
95 }
```