This file is copyright © 2006 Mark Jason Dominus. Unauthorized distribution in any medium is absolutely forbidden.

CGI Domain Adding Utility

I decided to present this program because the first thing I saw when I looked at it was:

```
11  #preset IPs for quick choice
12  my $server1ip = "127.0.0.1";
13  my $server2ip = "127.0.0.2";
14  my $server3ip = "127.0.0.3";
15  my $server4ip = "127.0.0.4";
16  #add your own here...
17  #my $server5ip = "127.0.0.5";
```

and this, of course, is a **family of variable names**, so we should try replacing it with an array, and see what happens. Lines 11--17 become:

```
#preset IPs for quick choice
my @serverip = ('127.0.0.1', '127.0.0.2', '127.0.0.3', '127.0.0.4',
                # add your own here
                );
```

and this:

```
35  if($server eq "server1"){$serverip = $server1ip;}
36  if($server eq "server2"){$serverip = $server2ip;}
37  if($server eq "server3"){$serverip = $server3ip;}
38  if($server eq "server4"){$serverip = $server4ip;}
39  #NOTE
40  #it is possible to add more shortcut hosts to this list;
```

```
41  #add another similar line here and define $serverNip at the start of the
42  #also add it in to the html file.
```

becomes this:

```
if ($server =~ /server(\d+)/) {
  $serverip = $serverip[$1];
}
```

Now if we need to change the program to add a new server, we only have to add it on one place, not two. This obviates the four-line comment.

Uniform Data Representations

Next I wondered about the tags `server1`, `server2`, and the rest. They are coming in from a CGI form. The comment even says:

```
40  #it is possible to add more shortcut hosts to this list;
42  #[C[also add it in to the html file.]C]
```

I imagine that the HTML looked something like this:

```
<input type="radio" name="server" value="server1">Server 1<br>
```

which generates a radio button for selecting server 1.

But then why bother with the symbolic tag names at all? Why not just let the HTML form invisibly translate the user-friendly server tags to the IP addresses that the program really wants?

```
<input type="radio" name="server" value="[C[130.91.6.1]C]">Server 1<br>
```

In short, use uniform data representations: don't represent a server with an arbitrary tag in the HTML file and by an address in the program; represent it by an address in both places.

One reason why you might not want to do this is for security reasons. The CGI form is under the control of the web user. They can post whatever data they want, even if it's not on the original form. In particular, they could make up an arbitrary address and post it to the program, which would then operate on it, even though that might not be appropriate. Using the tags would prevent this, because then the program could be written to operate only on an address for which a tag was defined. But this program wasn't written that way, because the form also has a widget something like this:

```
<input type="radio" name="server" value="other"> Other:
<input type="text" name="otherip"><br>
```

as evidenced by this code, which handles it:

```
} elsif ($server eq "other") {
  $serverip = $otherip
```

If the CGI form were changed to communicate addresses instead of tags, then this code:

```
if ($server =~ /server(\d+)/) {
```

```
  $serverip = $serverip[$1];
}
```

would change to this:

```
if ($server =~ /^\d+\.\d+\.\d+\.\d+$/) {
  $serverip = $server;
}
```

`elsif`

This line immediately follows the other checks on the form of `$server`:

```
if($server eq "other"){$serverip = $otherip;}
```

It's logically part of the same test; we are trying to find out what `$server` looks like. So the `if` should be an `elsif`:

```
if ($server =~ /server(\d+)/) {
  $serverip = $serverip[$1];
} [C[elsif]C] ($server eq "other") { $serverip = $otherip; }
```

`elsif` without `else`

Writing an `elsif` without an `else` is a red flag. Again, this does not mean it is always a mistake; only that it is often a mistake, and then when we do it, we should pause to reflect. When you find yourself writing `elsif` without `else`, ask yourself "What would happen if none of the conditions were true?"

In this case, it would mean that `$server` was somehow garbled, and `$serverip` would never be assigned, and would remain undefined. Later, the program would emit the output:

```
<h2>On Server yobgorgle with IP </h2>
```

and then it will cheerfully create a syntactically-incorrect zone file, without giving any indication that anything is wrong. The zone file could cause serious problems. This is something that our program ought to be checking for:

```
if ($server =~ /server(\d+)/) {
  $serverip = $serverip[$1];
} elsif ($server eq "other") {
  $serverip = $otherip;
} else {
  bail("Unrecognized 'server' parameter '$server'");
}
```

Scattered Code

I thought that `$otherip` was an example of **Variable Use Immediately Follows Assignment** flag, and I was congratulating myself on my cleverness, because it was hard to see; 38 lines of code separate the assignment and the use:

```
5  my $otherip = param("otherip");
```

```
44  if($server eq "other"){$serverip = $otherip;}
```

All of the `bail()` function and its fat wad of output is in between. (Why?)

But it turned out that I was wrong, because there was another use even farther down:

```
70  if(($server eq "other") && ![C[$otherip]C]){ bail "An IP must be specifi
```

I'm not precisely sure what leson to draw from this example; I have a sense that it is a common error. Certainly the repeated test is a tipoff that something is not right. But the real problem here is not the repeated test; it's that the code for dealing with `$otherip` is scattered all over the program. The repeated test might have been eaasier to see if it wasn't wandering in the desert like the twelve tribes of Israel. Try to **keep related code in one place**.

Here's one way to do that:

```
if($server eq "other") {
  $serverip = $otherip
    or bail "An IP must be specified if another server is specified!";
}
```

If the test of the assignment bothers you, then use this instead:

```
if($server eq "other") {
  bail "An IP must be specified if another server is specified!"
    unless $otherip;
  $serverip = $otherip
}
```

Similarly, we have:

```
 3  my $domain = param("domain");

69  if(!$domain){bail "You must specify a domain!";}
```

and the obvious fix is:

```
my $domain = param("domain")
 or bail "You must specify a domain!";
```

Mystery Constants

```
77  if(open(NAMEDCONF, "< $namedconf")){
78  flock(NAMEDCONF, 2) || bail "Couldn't flock $namedconf";
79  @namedlines = <NAMEDCONF>;
80  close(NAMEDCONF);
81  }else{bail("Couldn't open $namedconf!");}
```

There's a number 2 hiding on line 78. What's that for?

Well, *everyone* knows that that is the magic code for an exclusive lock, right?

Ha! At the very least, it is going to puzzle some maintenance programmer down the line.

The bigger problem with this sort of constant is that it is system dependent. It is in fact 2 on every system with which I am familiar, and one might almost be excused for believing that it will never change. But dozens of people over the years wrote code like this:

```
# XXX hardwired $PF_INET, $SOCK_STREAM, 'tcp'
# but who the heck would change these anyway? (:-)
$pf_inet = 2;
$sock_stream = 1;
$tcp_proto = 6;

...

unless (socket(S, $pf_inet, $sock_stream, $tcp_proto)) {
  ...
```

The comment says "but who the heck would change these anyway?" Who indeed? Sun Microsystems, that's who. One day they released an operating system on which the value of `TCP_PROTO` was not 6. In my archive of Usenet articles, I found about a hundred reports from people with this exact problem. The code, incidentally, is taken from the `lib/chat2.pl` library that was distributed with every release of Perl from 5.000 through 5.6.1.

Avoid using system-dependent magic numbers; Perl goes to a lot of trouble to find out the right numbers and to make them available to you. In this case, we should prefer this:

```
use Fcntl ':flock';

flock(NAMEDCONF, LOCK_EX) || bail "Couldn't flock $namedconf";
```

These are also easier for people to understand.

The Condition that Ate Michigan

This is a very small example of this red flag:

```
77  if(open(NAMEDCONF, "< $namedconf")){
78  flock(NAMEDCONF, LOCK_EX) || bail "Couldn't flock $namedconf";
79  @namedlines = <NAMEDCONF>;
80  close(NAMEDCONF);
81  }else{bail("Couldn't open $namedconf!");}
```

If we open up the condition, we get:

```
open(NAMEDCONF, "< $namedconf") || bail "Couldn't open $namedconf!"
flock(NAMEDCONF, LOCK_EX)       || bail "Couldn't flock $namedconf";
@namedlines = <NAMEDCONF>;
close(NAMEDCONF);
```

These error messages should describe the cause of the problem, so let's make sure they include `$!`:

```
open(NAMEDCONF, "< $namedconf") || bail "Couldn't open $namedconf: $!"
flock(NAMEDCONF, LOCK_EX)       || bail "Couldn't flock $namedconf: $!";
@namedlines = <NAMEDCONF>;
close(NAMEDCONF);
```

Use Sensible Variable Names

In the years I've been teaching this class, I've tried hard not to spend a lot of time talking about how to select variable names, because I think people talk about it too much and always say the same things, and that everyone is sick of hearing about it. But I just had to say something about this:

```
82  my $foooo;
83  my $baaar;
```

figure 6.1: `horror`

What the heck are these for, anyway? The key is line 99:

```
99  $filename = "$baaar/$foooo";
```

Uh, how about naming them `$file` and `$directory`? Would that be too crazy?

Data Extraction

`$foooo` and `$baaar` appeared in this block:

```
84  for($i=0; $i<=$#namedlines; $i++)
85  {
86        if( $namedlines[$i] =~ /^zone \"$domain\" {/ )
87        {
88        $namedlines[$i+2] =~ s/\s*file \"(\S*)\";/$1/;
89        $file = $namedlines[$i+2];
```

```
90        }
91
92        if( $namedlines[$i] =~ /directory \"\S*\";/ )
93        {
94        $namedlines[$i] =~ s/\s*directory \"(\S*)\"\s*;\s*/$1/;
95        $directory = $namedlines[$i];
96        }
97  }
98
```

This is a C-style `for` loop; the generic advice is to see if we can't use `for (@namedlines)` instead. In this case, we can't, because of the use of `$namedlines[$i+2]` to peek ahead. Fair enough.

But that peek ahead worries me. Specifically, what if the pattern match on line 86 succeeds, but the one on line 88 fails? And this is not an implausible scenario. The author is expecting the zone file to contain sections that look like this:

```
zone "land-5.com" {
        type master;
        file "zone/land-5.com";
};
```

But if someone inserts a blank line in there, the program will be hosed, and `$file` will be set to something totally nonsensical. Later that file wil be written to, and who knows what will happen next.

It's often okay to assume that your input wil be in a certain format, and to take advantage of that assumption to simplify the code. But the program is risky as it was originaly written. It's cheap and easy to reduce the risk:

```
if( $namedlines[$i] =~ /^zone \"$domain\" {/ )
{
$namedlines[$i+2] =~ s/\s*file \"(\S*)\";/$1/
  or bail "Couldn't locate 'file' line in zone section for '$domain'";
$file = $namedlines[$i+2];
}
```

If that second pattern didn't match, the program wasn't going to do anything useful anyway. It's better to have it die quickly than to go ahead and do who-knows-what.

The File Name

The lines we've just been discussing made me think hard about whether I really understood what was going on. It looks to me like the idea is to take a string like `file "fredsdomain";` and trim off the decorations so that it just says `fredsdomain`. But then I got woried, because it seemed simpler and more obvious to do it this way:

```
$namedlines[$i+2] =~ /\s*file \"(\S*)\";/;
$file = $1;
```

Then I wondered if maybe the input looked like this:

```
BLAH BLAH BLAH file "fredsdomain"; BLAH BLAH
```

in which case the idea would be to set `$file` to:

```
BLAH BLAH BLAH fredsdomain BLAH BLAH
```

But I don't *think* this is the case. I had to do a web search to look up the format of a zone file, and even then I was not sure. If the `BLAH BLAH BLAH` is not going to be there, as I suspect, then I suggest:

```
if( $namedlines[$i] =~ /^zone "$domain" {/ )
{
  ($file) = $namedlines[$i+2] =~ /file "(\S*)";/
    or bail "Couldn't locate 'file' line in zone section for '$domain'";
}
```

The Directory Name

```
92    if( $namedlines[$i] =~ /directory \"\S*\";/ )
93    {
94    $namedlines[$i] =~ s/\s*directory \"(\S*)\"\s*;\s*/$1/;
95    $baaar = $namedlines[$i];
96    }
```

Similarly, I suggest:

```
elsif ( $namedlines[$i] =~ /directory "(\S*)";/ )
{
  $directory = $1;
}
```

I also got rid of some unnecessary punctuation; you do not need to backslash quotation marks in regexes, as they are not special.

`elsif` Without `else`

Now my version has:

```
if( $namedlines[$i] =~ /^zone "$domain" {/ )
{
  ($file) = $namedlines[$i+2] =~ /file "(\S*)";/
    or bail "Couldn't locate 'file' line in zone section for '$domain'";
}
elsif ( $namedlines[$i] =~ /directory "(\S*)";/ )
{
  $directory = $1;
}
```

So we should ask:

What if none of the conditions are true?

In this case the answer is:

Nothing.

We want to ignore those lines, and that's what the program already does. So please please remember that just because something is a red flag, that does not mean that you have to take any action to address it other than to think about it. A red flag is not something that is wrong; it is something that is often wrong, and that requires attention. Please do not going around telling people that Dominus says you should never have `elsif` without `else`. Sominus only said that you should ask yourself in that case whether it would be appropriate to have `else`. In this case, no `else` is needed.

Missing Data

Having extracted the directory name and the filename from the `named.conf` file, we then do this:

```
99  $filename = "$baaar/$foooo";
```

or, for those of us not living on the Bizarro World:

```
$filename = "$directory/$file";
```

figure 6.2: `Bizarro`

It's possible that control could have escaped from the `for` loop without finding all the lines we needed; in that case, `$file` or `$directory` or both would still be unset, and `$filename` would be garbage. So add:

```
bail "Couldn't find information for domain '$domain' in named.conf"
  unless defined $file && defined $directory;
$filename = "$directory/$file";
```

The Condition that Ate Michigan (again)

The next block in the program has a 53-line-long `else` section:

```
102    if ( -f "$filename") {
103          bail("Error! $filename exists!!!\n\n");
104     }
```

```
105   else
106    {
107      print "\n Creating dns file ...";
     ...
157  }
```

I had some trouble figuring out which brace matched which, particularly because of the curious indentation and punctuation of line 148:

```
148  };
```

There's no need to bury all this stuff in the `else` block:

```
bail "Error! $filename exists!!!\n\n"
  if -f "$filename";

print "\n Creating dns file ...";
...
```

Creating a File Safely

The next chunk of the program deals with creating the new zone file:

```
bail "Error! $filename exists!!!\n\n"
    if -f "$filename";

print "\n Creating dns file ...";
my $ret = system("touch", "$filename");
if($ret){bail("Could not touch $filename!");}
open(DNSFILE, ">> $filename") or bail("Cannot open $filename!");
flock(DNSFILE, LOCK_EX) or bail("Cannot flock $filename, please try again!");
print DNSFILE<<END;
...
END
```

As a longtime Unix systems programmer, the code here seems very weird to me. For example, what is the `touch` call for? `touch` creates a file. But the `open(...>>...)` will create the file anyway, so the `touch` does not seem to be serving any purpose.

The `-f` test creates what is known as a *race condition*, in which two instances of the program that happen to be running at the same time might cause a disastrous and hard-to-reproduc error if things work out just right. Suppose the file does not exist, and two people submit the CGI form at almost the same time. Two instances of the program run, and both reach the `-f` test at approximately the same time. Then both `-f` tests return false, so both instances of the program conclude that it is safe to continue, and both of them go ahead an try to create the file at once. Perhaps that is not a big problem, but then what is the `-f` test for?

It seems to me that what the author wants out of all this is:

1. Bail if the file already exists
2. Otherwise, create it and insert the indicated contents
3. Avoid race conditions, such as two processes both "creating" the file at the same time

One way to accomplish all this is to make use of the OS features that are designed for the purpose:

```
use Fcntl 'O_CREAT', 'O_EXCL', 'O_APPEND', 'O_WRONLY';

sysopen DNSFILE, O_CREAT | O_EXCL | O_APPEND | O_WRONLY, 0666
    or bail "Could not create $filename: $!";
```

`O_CREAT` means "create the file if it doesn't already exist"; `O_EXCL` means "fail to open the file if it *does* already exist." If two processes both try to open with `O_EXCL`, Unix guarantees that at most one of them will succeed. Now the file locking is unnecessary, so we can write:

```
print DNSFILE<<END;
...
END

close DNSFILE;
```

Now, suppose you are *not* a longtime Unix systems programmer, and you don't know all that Unix systems programming stuff about `O_EXCL`. What could you have done differently?

The rule of thumb for avoiding race conditions is this: suppose you are trying to synchronize the processes on some condition *C*, say on the presence or absence of some file. Then the test to see whether *C* is true must be *simultaneous* with the change to *C*. In the original code, we had:

```
bail "Error! $filename exists!!!\n\n"
    if -f "$filename";                                  [C[TEST]C]

print "\n Creating dns file ...";
my $ret = system("touch", "$filename");              [C[CHANGE]C]
```

Here the test and the change are not simultaneous. In all such cases, you can get race conditions. If you want at most one process to succeed, the test to see if the file exists must be simultaneous with the creation of the file if it doesn't exist. That's what `O_EXCL` is all about. It's also what `flock` is for. With `flock`, the test to see if the file is already locked is simultaneous with the acquisition of the lock if it isn't already locked.

If you know about `flock` but not `O_EXCL`, you use `flock`. But you must obtain the lock *before* you try to create the file; otherwise, two processes might both try to create it at the same time. But you can't lock the file until after you create it, so it seems you are at an impasse.

The solution is to lock some other file instead, one that is not important:

```
sub start_critical_section {
  my $SEMAPHORE = /tmp/semaphore.adddomain";
  open SEM, ">", $SEMAPHORE
    or bail "Couldn't open semaphore file: $!";
  flock SEM, LOCK_EX or bail "Couldn't lock semaphore file: $!";
}

sub end_critical_section {
  close SEM;
}
```

Then, anytime you want to synchronize some action so that only one process can be performing it at a

time, you use:

```
start_critical_section();

only one process will be doing stuff in here
at any time

end_critical_section();
```

The final code looks like this:

```
start_critical_section();
bail "Error! $filename exists!!!\n\n"
  if -f "$filename";
open(DNSFILE, "> $filename")
  or bail("Cannot open $filename: $!");
print DNSFILE<<END;
...
END
close DNSFILE;
end_critical_section();
```

We can write to `DNSFILE` itself safely because we know that at most one process at a time can possibly have it open, any other processes will get temporarily stuck at `start_critical_section()` until `DNSFILE` is closed again.

Final Count

The program started out 68 lines long, and ended 61 lines long, not counting the big strings.

Usually I do better, but this program didn't have much excess code to begin with; the family of variable names at the top was deceiving. I saved some lines there, saved a few in eliminating repeated tests, and some in cleaning up the file locking code. But then I spent a lot of the gains, partly on adding extra error checking and the rest on the working critical section routines. I think this was code well-spent.
