

Script Teez: Automating secure remote backups	2
Script Teez: Uptiming your e-mail.....	4
Script Teez: Using PHP to spruce up your Web site.....	6
Simple file counter in PHP with MySQL.....	9
Script Teez: More power from Perl.....	11
Script Teez: Core removal.....	14
Script Teez: Simple search and replace	17

Script Teez: Automating secure remote backups

In this month's Script Teez, I'll look at a script that performs a backup to a remote system across the Internet. The script uses `rsync` and `ssh` to create an encrypted data tunnel so no one can sniff out your important stuff.

Script Teez

If you're an IT professional who relies on the power and flexibility of Linux, I'll present a Linux shell script each month that is sure to meet your demanding needs.

First, you'll need to have these programs installed in order for the script to work:

- The *expect* language
- *ssh*
- *rsync*

On the remote site, *ssh* and *rsync* will need to be installed as well. Once you've installed all of these programs and provided for any other dependencies, we can easily perform this backup. Here's the script you'll need to write:

```
#!/usr/bin/expect -f
log_user 0
spawn /usr/bin/rsync /etc -ar --delete --copy-unsafe-links -l -t -e ssh
user@remote.site.com:./
expect "password: "
send "secret\n"
log_user 1
interact
```

Line by line

As you can see, this is a very simple script. It takes advantage of *expect*, a language that provides interaction with your environment, similar to a PPP dial-in script. Let's walk through the script so you fully understand how it works.

The first line simply tells the shell to use `/usr/bin/expect` to run the script. The second line tells *expect* to turn off logging. This is handy if you're running the script in a *cron* job and don't want to receive e-mail containing the connection information.

The third line tells *expect* to spawn the *rsync* program. This allows *expect* to have some control over the program so you can interact with it. The command-line options basically tell *rsync* to back up the `/etc` directory tree and attempt to keep user/group/file permissions intact. This line also tells *rsync* to delete any files on the remote site that are no longer on the local site and to copy *symlinks*. In addition, this line tells *rsync* to use *ssh* to connect so that all of your transmitted data is encrypted. Finally, you're telling *rsync* to log in as username *user* at the remote site *remote.site.com*. The final `./` tells *rsync* to copy the files to the user's home directory.

The fourth line tells *expect* to wait for the password prompt that *ssh* gives. You're using *expect* to run this backup because there is no way to give *ssh* a password on the command line. It will accept passwords only at the password prompt.

The fifth line tells *expect* to send the password *secret* once the password prompt has been detected. You need to terminate what *expect* sends by using `\n` to simulate pressing the [Enter] key.

The sixth line turns logging back on so that the output of *rsync* can be recorded in the e-mail *cron* sends you.

Finally, the seventh line tells *expect* to turn on user interaction. Because *rsync* doesn't prompt for anything after it begins, this is a good way to release the backup from the control of *expect*.

One problem

The only problem with this script, of course, is the fact that the password for the user at the remote site is stored in the script. For this reason, I recommend storing the script in the */root* directory with file permissions of 700. Instead of making a *symlink* to the file for *cron*, write a *cron* entry to call the script in that location. This provides you with as much security as you can have with such an operation. The script should also be run as root so that you can back up your */home* directories as well. A user will never be able to back up another user's directories because of a lack of permissions.

And that's it! Enjoy!

Script Teez: Uptiming your e-mail

In this article, we will look at a Perl script that includes your current Linux uptime in your e-mail signature. Because it's a simple Perl script, you can use this for e-mail, news, or pretty much anything else that strikes your fancy. It makes use of a little program called *linux_logo*, which is available from [Project Glue](#) and is written by Vince Weaver.

You'll need to [download](#) *linux_logo* and install it to make use of the uptime features of this little script. It's basically a quick and dirty way to get uptime information in your e-mail signature without a lot of added complexity—which makes it great for people learning the basics of scripting in general and Perl in particular.

The following is the script you'll need to write. Save it as *uptime.pl*. Omit the line numbers at the beginning of each line. They are there simply for reference and are not actual parts of the script.

```
1: #!/usr/bin/perl
2:
3: $home = $ENV{"HOME"};
4:
5: # get current uptime
6: $uptime = `linux_logo -g _a _F "#U"`;
7: @uptime = split(/Uptime /,$uptime);
8: @disp = split(/minutes/, $uptime[1]);
9: $disp = "Current Linux uptime: $disp[0]minutes.";
10:
11: if ($#ARGV lt 0 or $#ARGV gt 1) {
12:   print "Usage: $^X sigfile\n";
13:   exit(1);
14: }
15:
16: open (SIG, "$ARGV[0]") or die "Can't open $ARGV[0]";
17: while (<SIG>) {
18:   $sig .= $_;
19: }
20:
21: print "$sig\n\n$disp\n";
22:
23: #open (SIG,">$home/.sig");
24: #print SIG "__ \n$sig\n\n$disp";
25: #close SIG;
26:
27: exit(0);
```

Line by line

NOTE: The (`) character is not a single quote but a *backquote*, which is on the tilde (~) key.

The first line of this script points to your Perl interpreter, which is usually */usr/bin/perl* or */usr/local/bin/perl*. The third line sets the value of the *\$home* variable to your current home directory. This way, any user on your system can use it without worrying about hard-coding the home directory.

The sixth line calls *linux_logo* with a few simple parameters. See the man page for *linux_logo* for the many other command-line options you can use. In this case, we are simply giving *linux_logo* the commands to provide system information only (no displaying the interesting ANSI penguin it generates by default) and to make the display ASCII instead of ANSI (no color). We also provide it with a format string of *#U*, which tells it to display the uptime only. Give it a try. On the command line, type:

```
linux_logo -g -a -F #U
```

and you'll see that the resulting output is something like:

```
Uptime 20 hours 57 minutes
```

Now, this isn't very interesting, so we're going to take that output, format it a bit to make it a little prettier, and then incorporate it into our signature. Keep in mind, however, that to use the output from another program, we must place the command within single back quotes (the character on the same key

as the tilde) with Perl. This will take the output of the command we place inside the back quotes and assign it to the variable, which in this case is `$uptime`.

Line 7 takes the value of `$uptime` and splits it into an array, which we call `@uptime`. (The `@` symbol references an array in Perl.) In this case, we are splitting the variable `$uptime` into two parts, consisting of the spaces preceding the word `Uptime` and the text after that word. We do this because `linux_logo` centers its display, and we don't want our text centered. We now have two variables: `$uptime[0]`, which is just blank spaces, and `$uptime[1]`, which contains the number of hours and minutes the system has been up.

Line 8 takes the value of `$uptime[1]` and splits it into another array, this time called `@disp`. We do this for the same reason we split `$uptime` back on line 7. We don't need the trailing spaces from `linux_logo`'s formatting, so we drop them and use the word `minute` as our split marker. We now have two variables: `$disp[0]`, which contains the number of hours and minutes (less the actual word `minutes` or `minute`), and `$disp[1]`, which will contain only spaces or the letter `s` and a series of blank spaces.

On line 9, we create a new variable called `$disp`, which contains the text `Current Linux uptime:` followed by the value of `$disp[0]`. Using the above output of `linux_logo`, `$disp[0]` would contain the words `20 hours 57`. Because there's a space at the end of the string, we don't put a space between `$disp[0]` and the word `minutes` when we create our variable. So, the value of:

```
$disp = `Current Linux uptime: $disp[0]minutes.`;
```

is:

```
Current Linux uptime: 20 hours 57 minutes.
```

Since this is supposed to be tagged onto the end of a signature, we have to call `uptime.pl` with a single argument—namely, the file to append the uptime information to. Line 11 checks to make sure that we pass only one argument to the file. If you were to pass no arguments, or more than one argument, line 12 displays the syntax required to run the program, and line 13 exits with an errorlevel of 1.

Line 16 tries to open the signature file and assigns it to the file handle `SIG`. Lines 17 through 19 read the file associated with the file handle `SIG` line by line until it reaches the End of File (EOF) marker. The signature is assigned, line by line, to the variable `$sig`, as seen on line 18.

Finally, on line 21, we print the signature, add two line feeds to give us a blank line between the signature and the uptime display (the `\n` character in the string is the newline character), and finally print the uptime display variable, `$disp`.

Lines 23 through 25 can be used to write the signature and uptime display to a specified file in the home directory called `~/sig`. This might be necessary for some e-mail or news clients that will not read the signature from `STDOUT`. Because line 21 prints the signature to `STDOUT` (or standard output), the program calling `uptime.pl` must be able to read `STDOUT` as input. If the program you use does not do this, you may choose to uncomment lines 23 through 25 and tell your e-mail or news client to use `~/sig` as the signature file. To keep the signature file from getting stale, you may choose to run `uptime.pl` in a `cron` job every five minutes so that it is accurate within five minutes.

Finally, line 27 exits the program with errorlevel 0, indicating that everything worked perfectly.

Obviously, using a client that can read `STDOUT` as the signature input will give you more accurate results. Because this script is so simple, you could probably even have it run as a `cron` job every minute without having any adverse results.

And now you know

And that's it! Now you know how to write a very simple Perl script to insert the current system uptime into your e-mail or news signatures. Not very useful but nice to use when you want to brag to friends how long your Linux system has been running. Enjoy!

With this method, `$page` will properly expand to `/home/httpd/html/subdir/mypage.php` when you include `header.php` in the directory one level up. See how simple it is if you want to move everything from `/home/httpd/html` to `/var/www`? Just one page needs to be changed, instead of 100.

Then, just include the following piece of code in your Web page wherever you want the last modified date to be displayed:

```
<? echo date("m/d/y", $mod); ?>
```

Of course, you can change the `date()` function to display the last modified date however you like. In this case, display it as `mm/dd/yy`.

You can do the same for the footer of your page by placing the information you want to be displayed on each page into the file `footer.php` and using the following as the last line of code in your Web pages:

```
<? include (?footer.php?); ?>
```

You now have dynamically included headers and footers for each page. If you want to change the copyright information on your site from one year to the next, you can easily accomplish this by making one change to `footer.php`. The resulting change is made globally across your site.

Reading and displaying filenames from a directory

Another thing you may wish to do on your Web page is list the contents of a directory for download. Let's take, for example, a directory of software packages you have available for download. These packages change on a relatively constant basis since you're a very enthusiastic programmer. You'd much rather spend your time programming than maintaining your Web site, so having an easy way to list the current contents of a directory and autogenerate an FTP link to the file is more to your liking than modifying the Web page each time you update a file.

This technique is relatively simple to implement with a little bit of work. I'll also throw a little twist into the mix: You want to make the visitors to your page aware of the last time the files were modified. So you're not only generating a list of FTP links based on the contents of a directory but also informing the user of when you last updated the file. With a little creative PHP code, this is very easy to accomplish. For this example, I'll assume that you release your packages in RPM format. Simply include the following in your Web page:

```
Downloadable as:<br />
<ul>
<?...$open = "/home/ftp/pub/software";
    $handle = opendir($open);
    while ($file = readdir($handle)) {
        if (ereg( ".rpm$", $file )) {
            if (ereg( ".src.rpm$", $file)) {
                $mod = filemtime("/home/ftp/pub/software/$file");
            }?>
            <li> <a href="ftp://ftp.mysite.org/pub/software/
            <? echo $file; ?>"><? echo $file; ?>
            </a><br />
            <? } else {
                continue;
            }
        }
    }
?>
</ul>
<hr /><br /><br />
<b>Files last modified:
<? echo date("D, M d Y H:i", $mod); ?>
</b><br /><br /><br />
```

Now, what does this elegant little piece of mixed HTML and PHP perform? The first thing you do is define the `$open` variable and point it to the local directory where the software is stored, which is `/home/ftp/pub/software` because `/home/ftp` is the root directory for your FTP server. The next thing you do is open the directory with the `opendir()` function and assign to it the file handle `$handle`.

Next, enter a `while()` loop. This loop reads the contents of the directory and terminates when there are no files left to read. You assign each file found to the `$file` variable. The next line, which is the first `if()` statement, performs a regular expression against `$file` to determine if it is a file that ends with `.rpm`. Remember, all of your software is released in RPM format, so you want to display only the RPM files on the site.

If the file is an RPM file, you do another regular expression match against it to see if it ends with `.src.rpm`, which is used to store the source code of the program. You *are* writing open source software, right? At any rate, if this matches, you assign the last modification date of the file to the variable `$mod`. After this, you display the link on the Web page. As you can see, it is a normal FTP link pointing to the FTP site `ftp.mysite.org` and the appropriate directory. You then echo the contents of the `$file` variable (which is your filename) to the link twice. The first time is to complete the `href` tag, and the second is to display the filename to the user.

Finally, if the file is not an RPM, you do nothing with it and continue your `while()` loop. And once you have listed all of the RPM files in the directory and exited the loop, you display a line of text that indicates what date the source RPM was last modified.

PHP is an extremely flexible and powerful Web scripting language that more and more people are incorporating daily into their Web sites. It is very fast, which makes it ideal for serving Web pages, as it does not create any noticeable delay from when a visitor requests a page to when the page is served to him or her. And as I've just illustrated, PHP can be a great time-saver. The time it takes to initially write the PHP code for your pages is minimal compared to the time you save in manually updating your Web pages with information you really shouldn't have to update. This article introduced you to two simple little scripts to help make your life a little easier, so take the time you save not updating your Web pages to relax and enjoy that coffee.

Simple file counter in PHP with MySQL

In this article, we're going to look at a very small and simple PHP script to track file downloads. Many sites offer files for download—documents, programs, pictures, or just about anything else they decide is worthy of sharing with others. Have you ever wanted to be able to track how many downloads a particular file (or any file you offer for download) receives? Using MySQL and a few lines of PHP code, you can do so easily.

What we're talking about is a counter and redirector script. Here's how it works: The user clicks on the file to download from your Web site. The link is actually a link to your redirector script, which is passed a variable to match against in the database, as well as the real link for the file.

For instance, my Web site, [Freezer Burn](#), offers a number of RPM files for download. The information for the RPMs is stored in a database. The database contains a few columns that provide a unique ID number, the path of the file (to do a timestamp check), and the name of the RPM, as well as a column to track the number of downloads a particular package receives. While we're not looking at a database containing this kind of information in this article, you can expand this technique to be more than just a simple file counter.

I can illustrate this best with a live example. Let's suppose that we want to download a file called *document.zip*. Instead of making a link like this:

```
<a href="ftp://mysite.com/document.zip">document.zip</a>
```

we would use one like this:

```
<a href="redirect.php?file=document.zip&go=ftp://mysite.com/document.zip"> document.zip</a>
```

This code passes two arguments to the *redirect.php* script. The first is *\$file*, which contains the name of the file and is used to match the file against our database entry. The second is *\$go*, which is the real URL to download the file.

For a very simple database that tracks only downloads of files, you'd need three columns. The first is a unique ID number, the second is the filename, and the third is the number of hits. Let's use a database with the name *counter* and use the following structure for a table we call *files*. Launch MySQL using

```
mysql -u root -p
```

Then, at the MySQL prompt, create the database and switch to it using

```
CREATE DATABASE counter;  
USE counter;
```

Now you can make the table itself using this:

```
CREATE TABLE files (id INT(4) UNSIGNED ZEROFILL NOT NULL PRIMARY KEY AUTO_INCREMENT,  
file VARCHAR(20) NOT NULL,  
hits INT(8) UNSIGNED);
```

Next, we come to the actual *redirect.php* script. Your database is complete and ready to be used with the following PHP file:

```
<?php  
if ($file) {  
    mysql_pconnect("localhost","user","")  
    or die("Unable to connect to SQL server");  
    $query = "SELECT * FROM files WHERE file = \"\$file\"";  
    $tmp = mysql_db_query(counter, "$query") or die("Select failed");  
    $filex = mysql_fetch_array($tmp);  
  
    $id = $filex['id'];  
  
    $query = "UPDATE files SET hits = hits + 1 WHERE (id=$id)";  
    $result = mysql_db_query(counter, "$query");  
    if (!$result) {  
        echo "There was an SQL error!<br />";  
        echo mysql_errno() .": " .mysql_error()."<br />";  
    } else {  
        Header("Location: $go");  
    }  
}
```

```
} else {  
  Header("Location: http://mysite.com/downloads.php");  
}  
?>
```

This script is extremely simple and works like a dream. It initially connects to the database and retrieves the ID number from the database that corresponds to the filename passed in the *\$file* variable. It then assigns the value of *\$filex['id']*, which is the retrieved ID number, to the variable *\$id*. For some reason, PHP doesn't like a MySQL query string that contains a variable like *\$filex['id']*, so we reassign it to a variable name it likes better.

Then we use the *SQL UPDATE* command to update the database's hit counter. We increment the value of hits by one each time *redirect.php* is called, using the SQL query itself to do the math. If there is an error, the script will display the SQL error number and description. If the update is successful, the user is redirected to the URL stored in the variable *\$go*, which is the true download location for the file.

As a safe measure, we make sure that if someone loads *redirect.php* without a valid *\$file* variable, we redirect them to the site's download page (or any other page you feel is appropriate).

From this point, you can use the information stored in the database however you like. The whole download process is entirely transparent to users. When they click on the filename, they will immediately be prompted with the download dialog box of their browser. The SQL transaction is so fast that it is virtually nonexistent.

As you can see, building a file counter is very easy, and you use a very small script to accomplish the task. You can even go so far as to have your download page pull the value of the hits column from the database and display it beside the filename so that your visitors can see how many downloads a particular file has received. Or you can keep the information entirely internal, and the user will never be the wiser.

Obviously, the script can be expanded to do more than simply count the number of downloads a file has received. With a little work, it can be used to deny downloads to certain IP addresses or domain names, to record the IP address of the person who downloads the file, and so forth. For this illustration, it is a simple file counter, but with a little creativity, it can become much more than that.

Script Teez: More power from Perl

In this Daily Feature, we're going to take another look at the power of Perl. Perl is probably one of the oldest and most-loved scripting languages around, and it's available for virtually every platform. Not so long ago, I had to look into a security issue with a popular Linux library called *ncurses*. The problem was that *setuid* root applications (programs that run as root regardless of which user executes the program) linked against *ncurses*. The tool to use to find out if a program is dynamically linked against *ncurses* is one called *ldd*. That program prints a list of all libraries dynamically linked against an executable file.

Of course, the problem existed only with *setuid* root applications, so I needed to find those first. Unfortunately, there are a high number of *setuid* applications, and for me to manually track down each one and run *ldd* against it would have taken hours. Instead of spending my time doing things manually, I chose to write a Perl script to do it all for me and print the results.

What I needed to do was simple yet time-consuming. I needed to find all *setuid* applications and then determine whether they were linked against *ncurses*, but I only wanted to see those applications that fit my search criteria. I didn't need a screen full of *setuid* applications. So a little bit of Perl programming solved the problem. Let me share with you the Perl script I wrote, which I called *checksuid*, and then explain what it does. While most people won't need to use the script as written, it illustrates the power of Perl and, with some changes, can be used for an infinite number of purposes.

The *checksuid* script looks like this.

```
#!/usr/bin/perl

# check suid programs to see if linked to ncurses

print "Gathering list of suid programs...
This may take a few minutes...\n";
@suid = `find / _uid 0 _perm +4000`;
chop(@suid);

print "Finished gathering... processing...\n";

foreach $bin (@suid) {
    $prob = 0;
    print "Checking suid root file: $bin\n";
    $temp = `ldd $bin|grep libform`;
    if ($temp) {
        print " $bin linked against libform\n";
        $prob = 1;
    }
    $temp = `ldd $bin|grep libmenu`;
    if ($temp) {
        print " $bin linked against libmenu\n";
        $prob = 1;
    }
    $temp = `ldd $bin|grep libncurses`;
    if ($temp) {
        print " $bin linked against libncurses\n";
        $prob = 1;
    }
    $temp = `ldd $bin|grep libpanel`;
    if ($temp) {
        print " $bin linked against libpanel\n";
        $prob = 1;
    }
    if ($prob == 1) {
        $problem .= "$bin ";
    }
}
```

```
print "Files linked against ncurses that are suid root:\n$problem";  
exit(0);
```

What it means

The first thing we do in this script is provide the location of our Perl program, which in most cases (under Linux) should be `/usr/bin/perl`. After that, we begin the search. Take a look at this portion of the script:

```
print "Gathering list of suid programs...  
This may take a few minutes...\n";  
@suid = `find / _uid 0 _perm +4000`;  
chop(@suid);
```

Our first step is to print a message to the screen, which lets us know what is happening. When it says it may take a few minutes, it means just that. We're searching every single file on the system, and it does take time.

The next step is to run the `find` program. What we are telling `find` to do here is to search every file under the root (`/`) file system with a permission of `4xxx`, where `4` is the numeric representation of the `setuid` bit. This means that files with a permission of `4700` or `4770` or `4777` (owner-execute, owner/group-execute, and all-execute) are found. We also tell it that we want it to match against a certain user ID. Since we're only interested in files that are `setuid` root, we use `uid 0`, which is always root. All the files that `find` returns are then placed into the array `@suid`. On the next line, we run the `chop()` command on the array, which removes the last character of every element in the array. Since the returned filename will contain the `\n` character (or newline character), we want to remove these from every element (returned file) in the array. After running `chop()` on the array, we're ready to use it.

Our next step is to create a `foreach` loop. The `foreach` loop will go through each element of the array, one at a time. When we run

```
foreach $bin (@suid) {
```

we're assigning the current element in the `@suid` array to the variable `$bin`. Every time the loop is run, `$bin` is the name of the next file to check. Because `@suid` contains only files that are owned by root and `suid`, we can then run the `ldd` program against the variable `$bin` because we know it fits our criteria.

Now we're looking only for files that are linked against `ncurses`. `ncurses` provides four libraries: `libform`, `libmenu`, `libncurses`, and `libpanel`. Because of this, we need to check for an instance of each library. But before doing so, we need to initialize a new variable that basically reports whether or not we have a problem file. We'll call this variable `$prob` and initialize it with a value of `0`, meaning "no problem." After we've done this, we want to check each file with `ldd` and interpret the results:

```
$temp = `ldd $bin|grep libform`;  
if ($temp) {  
    print " $bin linked against libform\n";  
    $prob = 1;  
}
```

At this point, we're checking for the library `libform`. We make a new variable called `$temp` in which we execute an external command. In this case, we're running `ldd` on the variable `$bin`, which is the name of the file to check. Since `ldd` will print every library the binary is dynamically linked against, we want to do a simple check to see if the library we're interested in is in the list. Thus, we pipe the output of the `ldd` command as input to the `grep` command, which in turn looks for the word `libform`. Because we're using `grep` in this fashion, `$temp` will be assigned a value only if `grep` returns true, meaning it found the library name we're searching for.

We then run the `if` command on the `$temp` variable. If it exists, then we've found the library we're looking for. So at this point we print a message to the screen indicating the file we checked and what library it is linked against. We then assign the `$prob` variable a value of `1`, meaning that `$prob` is true.

You see in the program that this command is repeated for each library we're interested in, so we include it four times. Once we've finished checking, we then need to check to see if `$prob` is true. If we don't find any libraries we're interested in, then `$prob` will be false (`0`). If even one library shows up, `$prob` will be true (`1`).

Since we also want to make a summary report once the script has finished, we want to keep track of which files are problematic. We do this using the following *if* statement:

```
if ($prob == 1) {  
  $problem .= "$bin ";  
}
```

What we do here is simple. If *\$prob* is true, then we assign the value of *\$bin*, which is the name of the file we're checking, to the variable *\$problem*. We use the *.=* assignment operator so that the name of the file is added to the list along with any other previous value of *\$problem*. If this is the first problem file, then *\$problem* will contain the name of the file. If this is the second, then *\$problem* will contain the names of the first and second files, separated by a space.

The *foreach* loop will continue to loop until our list of files is exhausted. Then the *foreach* loop will exit, and we can print our summary:

```
print "Files linked against ncurses that are suid root:\n$problem";
```

This will print to the screen the list of files that are problematic, which we stored in the *\$problem* variable.

If you were to run this program as is, the output of the script would look something like what's shown below. It isn't the entire output—just some interesting parts:

```
Gathering list of suid programs. This may take a few minutes.  
Finished gathering... processing...  
Checking suid root file: /bin/su  
Checking suid root file: /bin/ping  
Checking suid root file: /bin/mount  
...  
Checking suid root file: /usr/bin/xativ  
  /usr/bin/xativ linked against libncurses  
...  
Checking suid root file: /usr/bin/vmware  
Checking suid root file: /usr/bin/xatitv  
  /usr/bin/xatitv linked against libncurses  
...  
Checking suid root file: /usr/bin/xatitvc  
  /usr/bin/xatitvc linked against libncurses  
Checking suid root file: /usr/bin/cdrecord  
...  
Checking suid root file: /sbin/linuxconf  
  /sbin/linuxconf linked against libncurses  
Files linked against ncurses that are suid root:  
/usr/bin/xativ /usr/bin/xatitv /usr/bin/xatitvc /sbin/linuxconf
```

Conclusion

This script in its current form may not be overly useful to anyone other than someone performing security audits or programming functions. It can easily be tailored to search for other types of files that contain other content, however. It's the premise of the script that's important: You can spend a few minutes to write a simple Perl script to perform a function that could otherwise take hours.

Script Teez: Core removal

Ever find those annoying core files on your system? Those core files are the result of misbehaving programs exiting suddenly and leaving behind an informational dump that is only of use to a programmer and not much use to those of us doing everyday computing on the Linux platform. In fact, they are quite annoying and, if left unchecked, can claim a lot of disk space.

In this article, we use Perl once again to get rid of any core files on the computer in a nice, clean fashion with a relatively pleasing interface. We once again make use of the *find* program, which we use to locate the files.

This script is a little more elaborate and has the feel of a professional Perl program called *findcore*. The contents of *findcore* are as follows:

```
#!/usr/bin/perl
print "Looking for core files...\n";
#find command looks from / for real files (no dirs, symlinks, etc.)
#named core
@core = `find / -type f -name core 2>/dev/null`;
chop (@core);
$num = @core;

if (@core) {
if ($num == 1) {
print "Found $num core file... removing...\n";
} else {
print "Found $num core files... removing...\n";
}
foreach $file (@core) {
# if we have write (-w) permission for file, we can delete it
if (-w $file) {
system("rm -rf $file");
print "Deleted $file.\n";
} else {
print "You do not have permission to delete $file!\n";
}
} else {
print "No core files found!\n";
}
exit(0);
```

This script is relatively simple to write and use. Simply give it execute permissions and run it like this

```
chmod 755 findcore
./findcore
```

as any user. This program does some permission checking, so if you run it as a regular user and there is a core file found that can only be deleted by root (or another user), it will tell you.

The first step, as always, is to tell the system where to find the Perl interpreter. In most cases, it should be in */usr/bin/perl*. Then, we tell the user what we are doing. Since we're looking for core files, we tell them this with the print function.

The next step is to execute the *find* command and print the results to an array called *core*, which is represented by the variable *@core*. We use a different *find* command here than we did in the previous article, so let me explain the syntax. First, we search from the root directory, but we only look for files of a specific type; in this case we look for regular files. The reason for this is that we don't want to start deleting directories called *core* or symbolic links called *core*. Also, we don't want to delete any block device names, sockets, and so on. Core dumps result in a regular file, so this is the only file type we are interested in. That is what "-type f" stands for. (Read the *find* man pages to find out how to specify different file types.) Next, we only want files whose names match "core." All other files are of no interest to us at all. Finally, since *find* will tell us that we don't have permission to search in various directories and

print this information to the screen, we redirect the error messages to the bitbucket (*/dev/null*). We can't redirect everything to */dev/null*; otherwise, we will always have an empty array. Since we want the normal output that *find* gives us (the list of files matching our search pattern), we can only redirect STDERR (represented by the number 2) to */dev/null*. Thus, our final *find* command is:

```
find / -type f -name core 2>/dev/null
```

Next, we use the *chop* function on our array to remove all newline characters from any values of our array. We then assign the number of values in the array to the variable *\$num*. If *find* found two core files, there would be two elements in the array, and the value of *\$num* would be 2. If *find* doesn't find any core files, the value of *\$num* is 0.

Then, we do a test to see if *@core* contains any data. We could test to see if *\$num* is equal to 0, but testing *@core* itself is more efficient. If it does contain data, we perform another test to see if *\$num* is equal to 1. If it is, we print a string that in essence translates to "Found 1 core file...removing..." If *\$num* is not equal to 1 (and because of the previous test against *@core*, we know it is not 0), we then print "Found x core files...removing..." to the console, where x is the number of elements in the array, representing the number of files found. This gives us a nice and proper way to present what we found. We could have eliminated the test altogether and, if only one core file was found, have the program print "Found 1 core files...removing..." but that doesn't look as nice.

Now, we process the elements in the *@core* variable using the *foreach* loop. With the *foreach* loop, we assign the current element of *@core* to the variable *\$file* for the duration of the loop. Once again, we perform another test, but this time we check to see whether or not we have write permission to the core file as specified by the *\$file* variable. If we do not have write permission to the file, we can't delete it. We could eliminate this check and have the script print a bunch of "permission denied" errors to the screen, but we've gone to all this trouble to keep the interface clean, so we might as well continue with the trend.

In this case, we use the "-w" test, which is native to Perl, to see whether we can write to the file. If we can, we execute a system call to forcibly remove the file and then print "Deleted [filename]" to the screen. If we do not have write permission, we don't try to remove the file and simply print "You do not have permission to delete [filename]!" to the console.

The *foreach* loop will go through each element of the *@core* array, so it will process every core file that *find* reported. Once this is done, we exit with an exit value of 0, which means the program exited successfully.

Finally, if *@core* is empty, we print the string "No core files found!" to the console and also exit with an exit value of 0.

All about presentation

This script could have easily been a one-line BASH script if we didn't care about presentation. We do, however, care about presentation, and all we want to see on the screen is the important data. A long display of "permission denied" messages are of absolutely no interest to us, so we took a little bit of time to write a nice Perl wrapper around two basic commands: *find* and *rm*. The result is a nice professional-looking output from a relatively basic Perl program.

Once again, we see that Perl is a versatile tool and exceptionally useful for basic system administration. Those who think that Perl is only good for CGI scripts really don't understand how flexible the language can be for basic everyday use.

Conclusion

If you look at the last [Script Teez](#), you will realize that the basic fundamentals are the same. Both scripts have the same basic function: finding certain files. The actions that are run on the returned files are the differing factor here. In the last article, we simply reported on the found files, while in this article we remove the found files. The uses for this kind of script are infinite. You can modify the script to go through your system and remove a certain type of file, whether you want to remove a bunch of text files matching a certain file pattern or you want to remove all backup files. It's also simple enough that someone without a great understanding of Perl can make use of it for a variety of situations.

Script Teez: Simple search and replace

In this article, we are going to look at something a little different. Compared to some of the previous scripts we've written over the last few months, this one is perhaps the simplest and most useful of the bunch. I wrote this script about a year ago and use it on a regular basis, so I thought I would share it.

This is a simple bash script that makes use of the *sed* program. *Sed* is a stream editor, which means it takes input and transforms it based on user-specified criteria. *Sed* is a beast all unto itself, and it can be quite complicated. Fortunately, in its most basic form, *sed* is extremely easy to use and very useful. In this script, we will be using *sed* to search for one string and replace it with another.

Let's take a look at the script itself. I call this script *makenew*, and it looks something like this:

```
#!/bin/sh

list = ``ls -l *.php``

for file in $list
do
  sed ``s/DataBase/Database/g`` $file >t.1
  mv -f t.1 $file
done
```

It doesn't look like much, but I find it very useful, especially for Web work. What this script does is use as input a list of files in the current directory. In the above case, it will look for all the *.php files in the current directory. Using *ls -l* ensures we get only the filename itself and no extraneous information.

Then, we start a loop through all of the files found by the *ls* command. For each file, we execute a *sed* command on the file, which is represented by the *\$file* variable. In the above script, we want to change each occurrence of *DataBase* to *Database*. Since *sed* normally outputs the changed text to the screen, we redirect it to the temporary file *t.1*. Finally, we force a move of the temporary file with the changed string to the original file and then go through the loop again until there are no files left.

The script is meant to be edited manually each time you want to run it. There are really only two lines that need to be modified depending upon your circumstances. Obviously, you need to tell the *ls* command what files you want to modify. In the above example, we were looking for *.php files, but you might wish to edit .html or .txt files. You can use any standard *ls* argument here to list files.

The second command you'll change is the *sed* command itself. Since *sed* can be quite complicated, we'll simply look at its most basic form: a search and replace tool. The syntax you want to use for *sed* is: *s/string_to_find/string_to_replace_with/g*

In the above example, we searched for *DataBase* and replaced each occurrence with *Database*. You can do this with any kind of string; I typically use it for path changes. For instance, if I move my Web pages from */home/httpd/html* to */var/www/html* and make a lot of references to the path in my PHP code, I would use the following *sed* command in the *makenew* script:

```
sed ``s\/home\/httpd\/var\/www\/g`` $file >t.1
```

Because the forward slash (/) character has special meaning to *sed* (it separates the commands), you need to escape the forward slash with a backslash (\) if you want to use it in your string. In the above example, we are replacing all occurrences of */home/httpd* with */var/www*. Because there are forward slashes in our path, we have to use *VhomeVhttpd* and *VvarVwww* to make the proper changes.

Now, instead of manually editing any number of Web pages with that path reference, I simply run the script like this:

```
./makenew
```

and it is all taken care of automatically. By editing one file, I have saved myself from having to edit perhaps a dozen or more files. Be sure that you make the script executable prior to trying to run it by issuing the command:

```
chmod u+x makenew
```

And that's all there is to it! A very simplistic script but a definite time-saver for anyone who might have to edit multiple files to change the same string over and over again. If you are interested in learning about

the more advanced ways of using *sed*, be sure to read the man pages. Short books could easily be written on the many uses of *sed*, and the man page is a great place to start.

Vincent Danen, a native Canadian in Edmonton, Alberta, has been computing since the age of 10, and he's been using Linux for nearly two years. Prior to that, he used OS/2 exclusively for approximately four years. Vincent is a firm believer in the philosophy behind the Linux "revolution," and he attempts to contribute to the Linux cause in as many ways as possible—from his [Freezer Burn Web site](#) to building and submitting custom RPMs for the Linux Mandrake project. Vincent also has obtained his Linux Administrator certification from [Brainbench](#). He hopes to tackle the RHCE once it can be taken in Canada.