Getting More Out Of Apache (Part 2)

By icarus

Table of Contents

Tip O' The Iceberg

Last time, I showed you how to configure Apache to service multiple Web sites on the same physical server, and include server−side information in your Web page. But that's just the tip of the iceberg − Apache allows you to do a whole lot more, and over the next few pages, I'll be discussing the server's authentication and logging capabilities, together with a brief look at its unique URL re−writing module.

Keep reading!

Eyes Only

Security has always been a prime concern so far as the Internet is concerned; barely a week passes without media reports of security breaches at one Web site or another. If this is something you're concerned about (and you should be), you can set up Apache to protect confidential information on your Web site with a simple form of user authentication.

Apache's user authentication mechanism is based on the traditional username−password challenge mechanism. When the Web server receives a request for a directory or file that it knows to be a protected resource (aka "realm"), it responds by sending the client browser an authentication challenge. It is only after receiving a valid username and password back from the client browser that access is granted to the realm.

The concept is simple, and it works well; however, implementing it requires a little more work.

The simplest way to add protection to a specific directory is via the ".htaccess" file. In order to see how this works, create a file named ".htaccess" in the directory you wish to protect. Open the file in your favourite text editor and add the following lines to it:

```
# members table
AuthType Basic
AuthName "Top−Secret Information"
AuthUserFile /usr/local/apache/auth/mfre/users
require valid−user
```

The first two directives are pretty standard − the AuthType directive specifies the type of authentication (usually "Basic", although there is also a "Digest" type of authentication), while the AuthName directive specifies a name or description for the resource. This description will appear in the client browser when the user attempts to access the protected directory, so you should choose something descriptive.

The AuthUserFile directive specifies the location for the file containing a list of authorized users, together with their passwords. This file should *always* be placed outside the Web server root, in an area not accessible to a browser; if this is not done, anyone can download the file and view the information in it.

Finally, the "require valid−user" statement specifies the kinds of users that have access to this directory − in this case, it means that all valid users (read: users listed in the authorization file) have the ability to view the contents of the directory. You could further restrict the number of people allowed access by specifying user or group names − for example, the statement "require user joe beth" would only allow users "joe" and "beth" access to this area.

You should be aware, however, that the server will only read the ".htaccess" file if it is configured to do so. In order to confirm this, open up your main Apache configuration file, "httpd.conf", and look for the tags which reference your Web server root. These tags should look something like this:

```
...stuff...
```

```
AllowOverride All
...stuff...
```

The

```
AllowOverride All
```

directive tells the server that global configuration parameters can be overridden by local ones − the parameters in the per−directory ".htaccess" file.

Identity Check

If you've been paying attention, you'll have noticed that there's one thing missing − the authorization file itself. And Apache comes with its own little utility to create the file − it's called "htpasswd".

Switch to the directory specified in the AuthUserFile directive above, ad run the "htpasswd" command to create a file containing authorized users − you might see something like this:

```
$ htpasswd −c users joe
Adding password for joe.
New password:
Re−type new password:
```

And if you peek into the file "users", you'll see that the user has been added to the file − the garbage next to the user name is the password you just entered, in encrypted form.

```
$ cat users
joe:9DyNcHx.8JOp2
```

You can add as many users as you like using the method above (remember to omit the −c parameter, though, since that's only used when creating a file for the first time).

With everything in place, start up your browser and point it to the directory you just protected. The Web server should immediately pop up a dialog box asking for a username and password, and will only allow you to view the contents of the directory if you enter the correct values. Ain't that cool?

Grouping Things Together

In addition to user−level access, Apache also allows you to create different groups of individual users, and restrict access to protected resources based on group membership. For example, let's assume that I have two groups, "accounts" and "admin", and I would like to restrict access to a directory only to members of the "accounts" group.

The first thing to do is change the ".htaccess" file in the directory to read

```
AuthType Basic
AuthName "Top−Secret Information"
AuthUserFile /usr/local/apache/auth/mfre/users
AuthGroupFile /usr/local/apache/auth/mfre/groups
require group accounts
```

Next, you need to create the group memberships file, as specified in the AuthGroupFile directive above. This file is extremely simple − here's what it looks like:

```
$ cat groups
admin: bill harry
accounts: joe beth
```

Translation: users "joe" and "beth" are members of the group "accounts", while "bill" and "harry" are members of the group "admin".

Now, when you navigate back to the protected directory in your Web browser, Apache will again ask you for authorization; however, only members of the group "accounts" will be permitted to view the contents of the directory.

Finally, you can allow or deny access based on host names − the following lines would deny access to the protected resource for all requests from domains other than "melonfire.com"

```
order deny,allow
deny from all
allow from melonfire.com
```

Timmmmmmbbbbberrrr!

You may not know this, but Apache comes with some pretty impressive logging capabilities, which allow you to record demographic information about visitors to your Web site. And these logging capabilities can be customized to deliver exactly the information you need for later analysis.

Apache comes with two types of logs: there's the "access log", which tracks each and every request made to the Web server, and the "error log", which tracks internal server errors, missing file and the like. There are a number of configuration directives in the "httpd.conf" file which allow you to control Apache's default logging behaviour.

The ErrorLog directive specifies the location of the error log.

```
ErrorLog logs/error.log
```

The CustomLog directive specifies the location of the server's access log, together with a format for the log file (as defined in the LogFormat directive). The default setting is the Common Logfile Format, which places each request on a separate line; this format records the IP address, the date and time, the bytes sent, and the first line of the client request.

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
CustomLog logs/access.log common
```

The variables that you see in the LogFormat directive are server variables that return various identifiers for each client request − you can use these to create your own log format. More information about each variable does can be obtained from the Apache manual at http://httpd.apache.org/docs/mod/mod_log_config.html#customlog , or from the list below.

%h − requesting host

%b − bytes sent

%{VARNAME}e − value of environment variable VARNAME

%t − timestamp

%h − remote host

%T − time taken to serve request (seconds)

%U − URL requested

%s − return code for request

%{User−agent}I − remote user agent identifier

When Things Go Wrong

By default, Apache logs the IP address of each requesting client. If you'd prefer something a bit friendlier, you can have Apache log the host name of each client, rather than the IP address. This behaviour is controlled through the HostNameLookups directive − turn it on to enable name lookups.

```
HostnameLookups On
```

Be warned, however, that turning this feature on is likely to slow down the Web server, since it will have to perform a lookup for each distinct visitor to your Web page.

And finally, you can specify the type and severity of errors that are to be logged to the error log with the LogLevel directive. A number of error levels are available, ranging from "emerg" and "crit" (for severe errors) to "info" and "debug" (less−critical information).

This directive comes in particularly handy when you roll your own Apache server and things aren't working quite as advertised.

Just Rewrite It!

One of Apache's more powerful features − and also one of its least−known ones − is its URL rewriting module, which makes it possible to manipulate URLs using a rule−based rewriting engine in combination with regular expressions.

Apache's URL rewriting engine was originally developed by Ralf Engelschall in 1996. He handed it over to the Apache developers, who integrated it into the Web server, and it's been part and parcel of it since then. Note, however, that the engine needs to be compiled into the server to use it − so if you plan on trying out any of the examples below, you may need to re−compile your server with the mod_rewrite module first.

The basis of the mod_rewrite module is rulesets and conditions, which may be specified in either the main "httpd.conf" configuration file or the per−directory ".htaccess" file. When the server receives a HTTP request, the rewriting engine begins checking for a matching ruleset; if it finds a match, it substitutes the rewritten URL for the original one.

It's also possible to add conditions to specific rules − in this case, once a ruleset is matched, the conditions attached to it are checked, and URL substitution occurs if they are found to be valid.

Rewriting rules are specified with the RewriteRule directive, while conditions are specified with the RewriteCond directive. RewriteCond directives need to precede RewriteRule directive.

Putting It Into Practice

One of the simplest applications of the rewriting engine is also one of its most valuable − the ability to prevent Internet users from using images from your site on theirs. By carefully using the rewriting rules in combination with server variables, you can set things up so that other Web sites attempting to link to images on your site will not be granted access.

Here are the lines you need to add to your configuration file:

```
RewriteEngine On
RewriteCond %{HTTP_REFERER} !^$
RewriteRule .*\.jpg$ - [F]
```

The first directive is obvious − it turns the rewriting engine on. Once the engine is active, HTTP requests are scanned and matched against rules in the file.

The rules − there may be more than one, and they are interpreted in the order in which they appear − are specified via regular expressions, as in the example above. The first parameter following the RewriteRule directive is a pattern, while the second is the substitution pattern; you can also add special flags as a third parameter to invoke specific behaviour.

The rule above matches HTTP requests for images − files with the .jpg extension. Typically, you would replace these URLs with another string; however, I've used a hyphen to indicate that no substitution is to take place. Instead, I've used the [F] flag to have the server return a "403 Forbidden" result to the requesting client.

By itself, this is not enough − if you left it the way it was, every request for an image would be denied. It's therefore necessary to add a condition which checks whether the request is from another server or not. This can be done by checking the value of the HTTP_REFERER variable, which will usually not be empty if the request is coming from another server. The RewriteCond directive above checks the value of this variable, and activates the rule only when the HTTP_REFERER variable is not empty.

Another interesting application is using the rewriting engine to point your Web server's document root to a different physical location on your server's hard drive. For example, if you wanted all requests to the server to be served from the folder

```
/this/servers/new/root
```

rather than

```
/
```

you could use this simple rule:

```
RewriteEngine on
RewriteRule ^/$ /this/servers/new/root/ [R]
```

The [R] flag is used to indicate redirection.

This are just two simple examples which demonstrate the power of URL rewriting − it gets more complex as you get deeper into it. If you're really interested in find out what else you can do with the URL rewriting engine, take a look at Ralf Engelschall's Web site at http://www.engelschall.com/ , and at the Apache manual at http://www.apache.org/docs/mod/mod_rewrite.html

And that's about it from me for this week. See you soon!