SECURING APACHE

In this final part of the series, we will discover how to strengthen security in Apache by logging and other miscellaneous ways.

Configuring a system to be secure is indeed a key task, but it is also important to know that the configuration is working properly — and the only way to do so is through log analysis. Sensible logging helps detect performance problems well before they become apparent to users, and provides evidence of potential security problems. Maintaining logs is also useful for traffic analysis.

Apache can produce many types of logs, the two essential ones being the access log, where all requests are noted, and the error log, which is designed to log various informational and debug messages, plus every exceptional event that occurs. You, as Web master, have a limited amount of control over the logging of error conditions, but a great deal of control over the format and amount of information logged about request processing (access log). The server may log

activity information about a request in multiple formats in many log files, but it will only record a single copy of an error message.

One aspect of access logs that you should be aware of is that the log entry is formatted and written after the request has been completely processed. This means that the interval between the time a request begins and when it finishes may be long enough to make a difference. For example, if your log files are rotated while a particularly large file is being downloaded, the log entry for the request will appear in the new log file when the request completes, rather than in the old log file when the request was started. In contrast, an error message is written to the error log as soon as it is encountered. Let us look at these two scenarios, individually.

Access log

The access log is created and written to by the module mod_log_config, which is not a part of the core. To use the logging facility to its fullest, we need to first discuss the three configuration directives to manage request logging.

LogFormat

This directive specifies the format of the access log file and is known as Common Log Format (CLF). The default format is:

LogFormat "%h %l %u %t "%r" %>s %b" common

The first parameter is a format string indicating the information to be logged, and the format in which it should be written; the second parameter gives the format string a name. You can decipher the log format using the symbol table available at Apache website. Here, usingmod_logio, you can also measure the number of bytes transferred for every request. This feature allows hosting providers to put accurate billing mechanisms in place. The counting is done before SSL/TLS on input, and after SSL/TLS on output, so the numbers will correctly reflect any changes made by encryption.

Now that you are familiar with format strings, let's look at commonly used log formats:

common (the default) - %h %l %u %t "%r" %>s %b
combined - %h %l %u %t "%r" %>s %b "%{Referer}i" "%{User-Agent}i"
vcommon - %v %h %l %u %t "%r" %>s %b
vcombined - %v %h %l %u %t "%r" %>s %b "%{Referer}i" "%{User-Agent}i"

However, we can also create our own log format, but it is usually recommended to only use the above, as these are supported by log analysers.

TransferLog

This is the basic request logging directive, which creates an access log with the given filename:

```
TransferLog /var/www/logs/access_log
```

The filename can be given with an absolute path, as above. If a relative filename is supplied, Apache will create the full path by prepending the server home directory (e.g.,`/usr/local/apache`). By default, the TransferLog directive uses the Common Log Format (CLF), which logs every request on a single line with the information formatted. Here is an example of what such a line looks like:

```
81.137.203.242 - - [29/Jun/2004:14:36:04 +0100] "POST /upload.php HTTP/1.1" 200 3229
```

CustomLog

This directive is used to log requests to the server. The equivalent to the TransferLog usage described above looks like what is shown below:

```
CustomLog /var/www/logs/access_log custom
```

The first argument specifies the location to which the logs will be written. The second argument specifies what will be written to the log file. It can specify either a nickname defined by a previous LogFormat directive, or it can be an explicit format string as described in the log formats section. For example, the following two sets of directives have exactly the same effect:

```
# CustomLog with format nickname
LogFormat "%h %l %u %t "%r" %>s %b" common
CustomLog logs/access_log common


# CustomLog with explicit format string
CustomLog logs/access_log \"%h %l %u %t \"%r\" %>s %b\"
```

However, the first argument can also start with a pipe character, `|`, followed by the path to a program to receive the log information on its standard input. As by default, Apache uses the CLF, which does not record many request parameters. At the very least, you should change the configuration to use the combined format, which includes the `UserAgent` and the `Referer` fields.

Error logs

The Apache error log contains error messages and information about events unrelated to request serving. In short, the error log contains everything the access log doesn’t:

- Startup and shutdown messages
- Various informational messages
- Errors that occurred during request serving (i.e., status codes 400-503)
- Critical events
- Standard error output (stderr)

The format of the error log is fixed. Each line essentially contains only three fields: the time, the error level, and the message. In some rare cases, you can get raw data in the error log (no time or error level). Apache 2 adds the Referer information to 404 responses noted in the error log.
Error logs are created using the ErrorLog configuration directive. Standard file naming conventions apply here; a relative filename will be assumed to be located in the server main folder.

ErrorLog /var/www/logs/error_log
The directive can be configured globally or separately for each virtual host.
The LogLevel directive configures log granularity, and ensures more information than necessary is not in the log. Events that are on the specified level or higher will be written to the log file. The default setting is warn. On server startup, you will get a message similar to the following one:
[Mon Jul 05 12:26:27 2004] [notice] Apache/2.0.50 (Unix) DAV/2
PHP/4.3.4 configured -- resuming normal operations
You will see a message to log the shutdown of the server:

[Mon Jul 05 12:27:22 2004] [notice] caught SIGTERM, shutting down
Most other relevant events will find their way to the error log as well.

The Apache error log is good at telling you that something bad has happened, but it may not contain enough information to describe it. For example, since it does not contain information about the host where the error occurred, it is difficult to share one error log between virtual hosts.

There is a way to get error messages with more information using the mechanism of custom logging. Here is an example:

LogFormat "%h %l %u %t "%r" %>s %b "%{error-notes}n"" commone

CustomLog logs/super_error_log commone

Log rotation

Logs must be rotated on a regular basis, because accumulation of logs not only uses disk space, but also opening, closing and manipulating very large log files consumes system resources and will ultimately slow down the server. Thus, to easily handle log rotation, the`rotatelogs` utility is shipped with Apache, which uses piped logging, and rotates the file after a specified time period (given in seconds). For example:

```
CustomLog "|/usr/local/apache/bin/rotatelogs /var/www/logs/access_log 300"
custom
```

The above command rotates the log every five minutes. The `rotatelogs` utility appends the system time (in seconds) to the log name, to keep filenames unique. For the configuration directive given above, you will get filenames such as `access_log.1089207300`,`access_log.1089207600`, `access_log.1089207900`.
Given below are some more log directives that are useful while creating logs.

Logging cookies

The directives below will record all cookies sent to Apache by clients, and all the cookies Apache asks clients to set in their databases; this can be useful when debugging Web applications that use cookies. To log cookies received from the client, use the following code:

```
CustomLog logs/cookies_in.log "%{UNIQUE_ID}e %{Cookie}i"
CustomLog logs/cookies2_in.log "%{UNIQUE_ID}e %{Cookie2}i"
```

To log cookie values set and sent by the server to the client, use the code given below:

```
CustomLog logs/cookies_out.log "%{UNIQUE_ID}e %{Set-Cookie}o"
CustomLog logs/cookies2_out.log "%{UNIQUE_ID}e %{Set-Cookie2}o"
```

Note: Using the %{Set-Cookie}o format for debugging is not recommended if multiple cookies are (or may be) involved. Only the first one will be recorded in the log file.

Logging proxy requests

The directives shown below log requests that go through the proxy to a file that is different from the one from which the requests come directly to Apache. Here, use

the SetEnv directive to earmark those requests that came through the proxy server, in order to trigger conditional logging:

```
SetEnv is_proxied 1

CustomLog logs/proxy_log combined env=is_proxied
```

Logging the server IP address

Often, Apache is configured with virtual hosts with multiple addresses. Here, the directives to log the IP address of the virtual host that replied to the request will be as follows:

```
CustomLog logs/served-by.log "%{UNIQUE_ID}e %A"
```

Miscellaneous other ways to secure Apache

Use mod_parmguard

Definitely the most useful of the Apache modules, mod_parmguard (parameter guard) inspects incoming form submissions for abnormally-set parameters. The module includes a script that spiders your Web application, building up a profile of all forms in use. You can use this profile directly, or instead, tune it for better detection. For instance, the script might make sure that a parameter only got numeric values, but you could force those numeric values to be between 1 and 5. Configuring the module from Apache's point of view is extremely simple, because it has only three directives. The important part is the module's configuration file, which is based on XML, and is the heart of mod_parmguard. mod_parmguard has only two server-level directives (ParmguardConfFile and ParmguardTrace) and one location-wide directive (ParmguardEngine).

ParmguardConfFile sets the location for the module's configuration file. In this case, it would be/usr/local/apache1/conf/mod_parmguard.xml:

```
<IfModule mod_parmguard.c>
   ParmguardConfFile /usr/local/apache2/conf/mod_parmguard.xml
</IfModule>
```

You should have at least a minimal mod_parmguard.xml file set up. Here is an example:

```
<?xml version="1.0"?>
<!DOCTYPE parmguard SYSTEM "mod_parmguard.dtd">
<parmguard>
<!-- requested action when there is a mismatch -->
<global name="scan_all_parm" value="1"/>
```

<global name="illegal_parm_action" value="accept,log,setenv"/>
</parmguard>
For now, don't worry about the meaning of the options in this file. You must also remember to copy the file mod_parmguard.dtd to the same directory as mod_parmguard.xml, otherwise the XML parser will complain and won't let Apache start.
The ParamguardTrace directive is used in case you have a problem with the module, and you want to understand exactly what is going on. The only available option is debug. For example:

```
<IfModule mod_parmguard.c>
   ParmguardTrace debug
   ParmguardConfFile /usr/local/apache2/conf/mod_parmguard.xml
</IfModule>
```

The debug messages will be sent to Apache's error log.

The ParamguardEngine directive sets the location in which the engine actually is. This option must be placed in a <Location> directive in your httpd.conf (it will have no effect if placed in a directive). For example:

```
<Location /cgi-bin/>
   ParmguardEngine on
</Location>
```

Finally, as far as your httpd.conf is concerned, your module's configuration should be similar to what follows:

```
<IfModule mod_parmguard.c>
#ParmguardTrace debug
ParmguardConfFile /usr/local/apache2/conf/mod_parmguard.xml
</IfModule>


<Location /cgi-bin/>
#ParmguardEngine on
</Location>
```

Note: The directive ParmguardEngine is commented out for now, as no proper filter has been set yet.

Hide Apache version and other sensitive information

Attackers can use this information to their advantage when performing attacks. It also sends the message that you have left most defaults alone. There are two directives that you need to add, or edit in httpd.conf:

```
ServerSignature Off
```

ServerTokens Prod
The ServerSignature appears on the bottom of pages generated by Apache, such as 404 pages, directory listings, etc. The ServerTokens directive is used to determine what Apache will put in the Server HTTP response header. By setting it to Prod, it sets the HTTP response header as follows:
Server: Apache

Ensure that files outside Web root are not served

Apache should not provide access to any files outside its Web root. Assuming all your websites are placed under one directory (we will call this /web), you would set it up as follows:

```
<Directory />
  Order Deny,Allow
  Deny from all
  Options None
  AllowOverride None
</Directory>
<Directory /web>
  Order Allow,Deny
  Allow from all
</Directory>
```

Note: As we set Options None and AllowOverride None, this will turn off all options and overrides for the server. You now have to add them explicitly for each directory that requires an Option or Override.

Do not allow Apache to follow symbolic links

This again, can be done using the Options directive, inside a Directory tag.
Set Options to eitherNone or -FollowSymLinks.

Only give the root user read access to Apache's config and binaries

Assuming your Apache installation is located at /usr/local/apache, this can be done as follows:

```
chown -R root:root /usr/local/apache
chmod -R o-rwx /usr/local/apache
```

We will be back with other interesting articles on security and hacking.

Always remember: Know hacking, but no hacking.

Source : http://www.opensourceforu.com/2011/09/securing-apache-part-11-logs-et-al/