# Compiling, installing and configuring Apache and PHP on Linux


## Elliot Smith, moochlabs.com


## Table of Contents

# Introduction

This document outlines how to compile, install, and configure Apache and PHP on Linux. It is not a complete manual to the process, but goes through the process step by step, explaining the decisions to be made along the way.

We are working towards the following scenario:

- A secure, custom built and configured Apache web server with support for PHP 5 (including the MySQL and GD extensions) plus virtual hosts
- SSL support for our main website
- A default (package managed) MySQL installation, accessible to the Apache server
- Some PHP scripts to prove we can connect to the MySQL server from PHP, and that we can use the GD graphics toolkit
- A layout for virtual hosts: we're going to assume one client, with their own website at jelica.com
- A user account for the virtual host, isolated from the main Apache configuration, allowing the user to login and edit their website

Note that I wrote these instructions based on Ubuntu, but they should be portable to other Linux distributions. In particular, I have outlined Fedora-specific issues, as the materials were written for a training course run using machines installed with Fedora.

# Compiling Apache

## *Pre-compilation decisions*

Which version of Apache?

- 1.x
  Has been around for years, and is a known quantity. A safe choice.
- 2.x
  Code is much improved, and many of the modules have been revamped. Configuration is also more consistent, and the format for directives improved. However, some people have reservations about using it. Although it is possible to run in a hybrid multi-process/multi-thread mode (using the worker MPM), many of the libraries you're likely to use with it may not be (e.g. PHP extensions). However, under normal conditions (i.e. up to tens of thousands of hits per day, rather than millions), this version of Apache is likely to be a better solution than Apache 1.x.

Binary or source?

- Source = more control; you can patch when you want; you can add features when you like
- Binary = easier to manage; automatic updates; less control
  - via package management tool (using individual components), e.g. Apt on Debian, RPM on Fedora
  - via a pre-packaged stack containing all components, e.g. XAMPP (http://apachefriends.org/en/xampp.html) - also gives some of the advantages of a source

installation, as you can compile new modules into it

- via a pre-packaged stack, with optional certification and support, e.g. SpikeSource (http://www.spikesource.com/downloads.html), Sourcelabs (http://sourcelabs.com/?page=software&sub=amp)

We'll do it from source, using version 2.2

# Preparation

Preparing the machine you're going to install on

- gcc
- OpenSSL
- OpenSSL development headers (libssl-dev on Ubuntu)
- ntpdate to ensure server time is accurate
- Perl 5 – allows you to use some of the support scripts like apxs (for building and installing shared modules)

Download the source and check the archive's integrity using md5sum like this:

```
root@lily:/home/ell/download# md5sum httpd-2.2.2.tar.bz2
9c759a9744436de6a6aa2ddbc49d6e81  httpd-2.2.2.tar.bz2
```

Compare the string on the left to the MD5 hash listed on the Apache download site. They should match. If they don't, the download has been corrupted, so do it again.

## Preparation on Fedora

On Fedora, I found I needed to install the following via "Add/Remove Software":

- Development > Development Libraries + Development Tools + GNOME Software Development

# Compiling

Unpack the tarball

Need to get apr up and running first:

cd httpd-2.2.2/srclib/apr

./configure --prefix=/opt/apache-apr

make

make install

Then apr-util:

cd httpd-2.2.2/srclib/apr-util

./configure --prefix=/opt/apache-apr-util --with-apr=/opt/apache-apr

make

make install

Then Apache:

cd httpd-2.2.2

./configure --prefix=/opt/apache --with-apr=/opt/apache-apr --with-apr-util=/opt/apache-apr-util

make

make install

Test:

/opt/apache/bin/apachectl start (as root)

NB you need to be root if the port Apache listens on (Listen directive) is below 1024; default is port 80

Test by visiting http://localhost/ in a web browser

# *Controlling Apache*

ps to see the processes Apache starts

When Apache starts, it establishes a parent process as the original user (e.g. root in our case); it then spawns child processes to handle requests. The number of children is configurable (see later).

The PID file stores the ID of the parent process. It can be sent a variety of standard POSIX signals to control it directly; or (better) it can be controlled through the apachectl script.

The files in the log directory are the default Apache logs, as specified by the auto-generated config. file. error_log is useful for debugging, and at the moment contains start/stop info.; access_log records requests served.

The apachectl script takes a variety of switches

**start** = start the parent process

**stop** (TERM signal) = tell the parent to kill its children; it does this immediately; then once they've exited, the parent kills itself

**graceful** (USR1 signal) = instruct the parent process to advise the children to exit; they allow all requests being served to complete; then they stop; then the parent stops; then the parent restarts itself; the parent process then starts new children with the latest version of the configuration file

**graceful-stop** (WINCH signal) = as graceful, but no restart after everything stops

**restart** (HUP signal) = this restarts its children (as in TERM), but doesn't stop the parent process; the parent process just rereads its configuration file and carries on running

**status** = show short status report (NB this needs lynx installed to work, and mod_status to be enabled)

**configtest** = test whether the config. file is readable and correctly formatted

# Modules

Modules add extra functionality to Apache. Their functionality is managed via Apache configuration directives; and each module makes different directives available.

Static- vs. dynamically-loaded modules?

- Static = whole server + modules in one binary; slightly faster; harder to compromise as you can't just link new modules into it; must recompile whole thing each time you update; uses more memory
- Dynamic: you need to have mod_so enabled (NB mod_perl should not be compiled as a shared module, according to http://www.faqs.org/docs/apache-compile/apache.html)

We'll do as many as we can as dynamic modules, while keeping the core static

To see the list of modules compiled into the httpd binary:

```
/opt/apache/bin/httpd -l
```

Here's what I got:

core.c (yes - essential for the server to operate)

mod_authn_file.c (yes - essential for Basic authentication)

mod_authn_default.c (yes - essential for authentication)

mod_authz_host.c (yes - authorization by hostname/IP)

mod_authz_groupfile.c (yes - authorization by groups defined in a file)

mod_authz_user.c (yes - authorization by users defined in a file)

mod_authz_default.c (yes - essential for authorization)

mod_auth_basic.c (yes - support for Basic authentication)

mod_include.c (no - unless you need server-side includes)

mod_filter.c (no - provides filtering of resources before they are returned in the response, e.g. zipping the response body, downsampling every image sent back from the server)

mod_log_config.c (yes - allows customisation of log output)

mod_env.c (no - unless need to set and clear environment variables for use with CGI scripts - e.g. essential if running Ruby on Rails applications with FastCGI)

mod_setenvif.c (yes - supports a lot of other modules)

prefork.c (yes)

http_core.c (yes)

mod_mime.c (yes - allows Apache to correctly deliver content based on MIME type)

mod_status.c (no - shows server status page)

mod_autoindex.c (no - unless you want directory indexes to be shown for directories with no index file)

mod_asis.c (no - used to send a file without appending response headers to it - so you could have a file which contains a whole HTTP response, including headers)

mod_cgi.c (no - unless you want CGI script support)

mod_negotiation.c (no - it provides a method for negotiating the best content type to suit the client's capabilities)

mod_dir.c (yes - controls the DirectoryIndex directive, used to set the default file to serve for a directory, e.g. index.php)

mod_actions.c (no - triggers CGI scripts based on the MIME type of a resource requested - e.g. all requests for image/jpeg are handed off to a specific CGI script)

mod_userdir.c (no - unless you want ~/public_html directories for user home sites)

mod_alias.c (yes - handles aliasing of URLs to directories)

mod_so.c (yes - shared object support for dynamic extension loading)

## Disabling modules

Any modules we want turned off have to be explictly disabled with this syntax:

--disable-MODULE

For our purposes:

--disable-userdir

--disable-actions

--disable-negotiation

--disable-cgi

--disable-asis

--disable-autoindex

--disable-status

--disable-env

--disable-filter

--disable-include

BUT we can also remove the remaining modules and make them dynamically-loaded:

--disable-mod_authn_file

--disable-mod_authn_default

--disable-mod_authz_host

--disable-mod_authz_groupfile

--disable-mod_authz_user

--disable-mod_authz_default

--disable-mod_auth_basic

--disable-mod_log_config

--disable-mod_mime

--disable-mod_dir

--disable-mod_alias

Note we didn't disable a few of the modules, as we do want them statically compiled (e.g. mod_so, which enables shared modules to be loaded)

### Enabling modules

Extra modules we want:

ssl (support for SSL - we'll put this in statically)

setenvif (set environmental variables conditional upon modules being loaded)

headers (enable modification of request/response headers)

rewrite (for rewriting requests - used for search-engine friendly URLs, for example)

deflate (for zipping content before it is sent to client [useful if client supported gzipped streams, e.g. Firefox])

cgi (for running CGI scripts)

The typical method (the one we'll use) is to use shared modules rather than static ones

We do this by adding this option to ./configure, with the names of the modules we want to enable:

--enable-mods-shared='setenvif headers rewrite deflate cgi'

But we will enable SSL as a static module, to ensure it is always used and to minimise the possibility of the library being trojaned.

--enable-ssl

We can also add back in the modules which were previously statically-compiled but which we are converting to dynamically-loaded modules:

--enable-mods-shared='authn_file authn_default authz_host authz_groupfile authz_user authz_default auth_basic log_config mime dir alias'

## *Other configure options*

If you want to be able to use apxs, it's a good idea to specify the path to Perl explicitly (just in case multiple versions are installed):

--with-perl=<path to perl executable>

As we have turned on ssl, best to explicitly set where OpenSSL is installed:

--with-ssl=<path to openssl include directory, e.g. /usr/include/openssl>

Full list of options to configure:

http://httpd.apache.org/docs/2.2/programs/configure.html

## *Other useful modules we're not using*

Here are some modules we're missing out, but which can be very useful:

- mod_dav (WebDAV support)
- mod_ldap (base module to support other modules, e.g LDAP authentication modules)
- mod_proxy (use Apache as a proxy to other servers)

- mod_proxy_balancer (for load balancing)
- mod_cache (cache local or proxied content)
- mod_vhost_alias (automatic mapping of URLs onto virtual hosts)

## Which Multi-Processing Module?

prefork is the default for Linux - stable, tolerant of dodgy module code (one process at a time handles each connection)

worker is more lightweight, but less tolerant (uses multiple child processes, plus each child has multiple threads - each thread handles one connection)

prefork is the recommended MPM to use if you intend to run PHP as a module (see http://www.php.net/manual/en/faq.installation.php#faq.installation.apache2); however, if you intend to use FastCGI or similar to run PHP, the worker MPM is stable.

To enable worker instead of prefork on Linux add the following configure option:

```
--with-mpm=worker
```

## Our uber configure command

Putting all of this together gives us our master configure command:

```
./configure --prefix=/opt/apache --with-apr=/opt/apache-apr --with-apr-util=/opt/apache-apr-util --with-perl=/usr/bin/perl --with-ssl=/usr/include/openssl --disable-userdir --disable-actions --disable-negotiation --disable-cgi --disable-asis --disable-autoindex --disable-status --disable-env --disable-filter --disable-include --disable-mod_authn_file --disable-mod_authn_default --disable-mod_authz_host --disable-mod_authz_groupfile --disable-mod_authz_user --disable-mod_authz_default --disable-mod_auth_basic --disable-mod_log_config --disable-mod_mime --disable-mod_dir --disable-mod_alias --enable-mods-shared='cgi setenvif headers rewrite deflate authn_file authn_default authz_host authz_groupfile authz_user authz_default auth_basic log_config mime dir alias' --enable-ssl
```

It would be a good idea to put this into a script, so you have it available each time you recompile Apache.

Remember that once we've run configure, we then need to do:

```
make
make install
```

This performs the compilation (according to our configuration) and installs the binaries into the appropriate location (under /opt/apache).

## Recompiling

Recompiling a new version of Apache (given an old version already exists) isn't too arduous. There are several things we might want to do:

1. Upgrade Apache as a whole (e.g. moving from version 2.2.55 to 2.2.57)
2. Compile modules statically into the httpd binary (either new ones or existing shared ones we want to move into the core httpd binary)
3. Compile new shared modules (either completely new ones or existing statically-compiled ones)

See http://httpd.apache.org/docs/2.2/install.html for more details. Outlines of each process are given below.

## 1. Upgrading the main httpd binary

You can only do this for minor version number changes, e.g. version 2.2.0 to 2.2.1; you can't do it to go between major version number changes, e.g. 2.0 to 2.2.

If you are upgrading, it's worth doing it alongside your existing installation. You could do this by changing the --prefix option to configure, so that the new version ends up in a different directory; and setting a different Listen directive inside the new httpd.conf file so your new version runs on a different port. One you're happy, you can re-run configure with the correct --prefix setting.

Here's the procedure:

1. Download the new source distribution and unpack it
2. Copy the *config.nice* file from your old source tree for Apache into the top of the new source tree. This file is basically a script which will replay all the configure options you used to build the old version.
3. Run the following commands:

   ```
   ./config.nice
   make
   make install
   ```

The Apache make file will not overwrite existing files on the server like configuration files (httpd.conf) or files which have changed. But it will overwrite the httpd binary and any modules which *have* changed.

## 2. Compiling modules statically into the main Apache binary

Let's say we have mod_ssl compiled as a shared module, and want to recompile our httpd binary to statically include it instead. We can do this as follows:

1. Pass an edited set of options to the ./configure script. For example, let's say we had SSL compiled as a shared module (a fragment of our configure options lines):

   ```
   ./configure --enable-ssl=shared ...
   ```

   Change this to compile the module statically instead:

   ```
   ./configure --enable-ssl ...
   ```

2. make

   The make command rebuilds the httpd binary (plus any other files which have changed as a result of our reconfiguration)

3. Manually copy the new httpd binary (in the root of the build directory) into our existing

Apache configuration, i.e.

cp ./httpd /opt/apache/bin/

4. Reset the permissions on the new binary (see later)
5. Remember to remove any LoadModule lines for the old shared version of the module, so that the statically-compiled module is used instead.
6. (Optional) Remove the shared module from the modules directory, as it is no longer being loaded.

We could follow the same approach to enable a *new* static module in the httpd binary (rather than move a module from being dynamic to static).

Alternatively, we could recompile, then use `make install` to overwrite our installation with any changed files (see above).

## 3. Compiling new shared modules

We could do this to either add a completely new shared module, or to move a static module to being a shared module.

The apxs tool can be used to add new shared modules into an existing Apache installation. The procedure may vary slightly from module to module, but for the ones which are part of the core Apache distribution it follows this pattern:

1. Locate the module directory (in the source tree, under modules). The modules are arranged into groups, e.g. proxy for modules which handle proxying functions, mappers for mapping modules like mod_rewrite. What we're looking for is the appropriate .c file for the module.
2. Run the apxs command with the -c (compile) and -i (install) flags, e.g.

   /opt/apache/bin/apxs -c -i -a mod_rewrite.c

   This compiles up the new module binary (.so file) and deposits it into /opt/apache/modules.
3. Note that the -a switch to apxs automatically adds a LoadModule line to httpd.conf. If you don't use this switch, you will need to manually add the LoadModule directive to httpd.conf yourself, something like this:

   ```
   LoadModule rewrite_module modules/mod_rewrite.so
   ```

If we want to move a static module to become a shared module, we will need to recompile the httpd binary as well, and exclude the old static module (see instructions above).

We can demonstrate how this works by compiling a simple module like mod_echo. This turns the Apache server into an echo server which repeats back whatever you send to it.

1. cd <Apache source root>/modules/echo
2. /opt/apache/bin/apxs -c -i mod_echo.c
3. Edit /opt/apache/conf/httpd.conf and add these lines:

   ```
   LoadModule echo_module modules/mod_echo.so
   ProtocolEcho On
   ```
4. Restart Apache

5. Test the module has loaded correctly using telnet:

   ```
   telnet localhost 80
   ```

   Type some commands, and they should be echoed back to you. This is Apache acting as an echo server, using its newly-compiled echo module.

The beauty of Apache's modularity is that it is equally easy to remove a shared module. We can simply remove the LoadModule directive in httpd.conf; and we could additionally remove the .so file itself to be extra safe.

# Patching

Occasionally, between releases of Apache versions, official patches may be released for the current version. These patches will typically implement important security updates which are too vital to wait until the next full release. They are fairly rare, but you should check for applicable patches before compiling.

To get the patches, go to the source download directory on one of the mirror sites, via the Apache Downloads link. Inside the main distribution directory is a patches directory, e.g.

http://www.mirrorservice.org/sites/ftp.apache.org/httpd/patches/

This contains a series of directories with names in this format:

`apply_to_2.2.0/`

Inside these directories are a series of patches for each released version of Apache. To apply a patch:

1. Download the patch file
2. Place it in the source directory for Apache
3. Apply it to your source with:
   `patch -s < file.patch`
   where file.patch is the name of the patch file you downloaded
4. configure/make/make install (see the Upgrading section above for more details about the effect of this)

# Configuring Apache

## *Default configuration*

The default configuration for our compiled Apache is in /opt/apache/conf/httpd.conf. There are other useful files containing sample configuration "fragments" in the /opt/apache/conf/extra directory. They can be used for reference or pulled into your main config. file as they are, with little modification.

## *Viewing all loaded modules*

To show all loaded modules (including dynamically-loaded modules) once the server is running:

```
/opt/apache/bin/httpd -M
```

Which outputs:

```
Loaded Modules:
 core_module (static)
 mpm_prefork_module (static)
 http_module (static)
 so_module (static)
 authn_file_module (shared)
 authn_default_module (shared)
 authz_host_module (shared)
 authz_groupfile_module (shared)
 authz_user_module (shared)
 authz_default_module (shared)
 auth_basic_module (shared)
 deflate_module (shared)
 log_config_module (shared)
 mime_magic_module (shared)
 headers_module (shared)
 setenvif_module (shared)
 ssl_module (shared)
 mime_module (shared)
 dir_module (shared)
 alias_module (shared)
 rewrite_module (shared)
Syntax OK
```

NB this also checks the syntax of the config. file; can do this without displaying modules using:

/opt/apache/bin/httpd -t

## Initial configuration

We have the option to use a single config. file OR spread it out over multiple files. Pros and cons:

- Single = everything in one place; difficult to manage with lots of vhosts
- Multiple = clear separation of different aspects of configuration

We'll use a single file for the main config.; plus a separate file for the SSL config., and one for each virtual host.

We are also writing a config. file which only works for our compiled version of Apache. The default generated file provided as an example in the Apache distribution contains a variety of conditional statements. These apply different configuration directives depending on the underlying operating system, but we are going to dispense with these as much as possible to get a streamlined configuration file.

Start with a blank config file in

/opt/apache/conf/httpd.conf

Then add:

```
# base of the web server install
ServerRoot /opt/apache
# name of the web server (can help prevent
# startup problems)
ServerName localhost
# email address of the administrator
# (shown in error messages)
ServerAdmin ell@localhost
# location of the root of the web server document tree
DocumentRoot /var/www/htdocs
# path to the process ID (PID) file, which
# stores the ID of the main Apache process
PidFile /var/run/apache/httpd.pid
# which port to listen on
Listen 80
# do not resolve client IP addresses to names (reduces overhead)
```

```
HostNameLookups Off
# effective user and group
User apache
Group apache
```

We will need to create the "effective user" under which Apache will run on the system. Apache will run with the permissions of this user and group:

groupadd apache

useradd apache -g apache -d /dev/null -s /bin/false

(-d = home directory, -g = main group, -s = shell)

And we'll need to create the appropriate directories:

mkdir /var/www/htdocs (for the resources Apache will serve)

mkdir /var/run/apache (for the process file)

mkdir /var/log/apache (for logs)

There are also several directories lying around which we can safely remove:

rm -Rf /opt/apache/manual (the Apache manual)

rm -Rf /opt/apache/cgi-bin (we'll put the cgi-bin into individual host configurations)

rm -Rf /opt/apache/icons (icons used when listing the content of directories - we're not going to be doing this anyway, so we may as well remove them)

It's necessary to keep the default logs directory, even though we're not using it directly, as some modules use it to store files, while not providing a directive to customise the log directory location.

If it's useful you can also keep the man directory. You can access the files in this directory using the man command like this:

man ./htdigest.1

for example, if you need to find out more about the commands Apache provides.

We can now try restarting to ensure our config. works:

/opt/apache/bin/apachectl restart

We won't be able to see our site anymore, but at least we should be able to see if the server starts OK.

## *Starting/stopping automatically*

Symlink into appropriate run-level

On Ubuntu, I suggest run-level 2

ln -s /opt/apache/bin/apachectl /etc/rc2.d/S85apache

ln -s /opt/apache/bin/apachectl /etc/rc2.d/K20apache

You also need to make sure that the network is up and the hostname set before you start the Apache server, so a high number like 85 is suitable.

### Starting/stopping automatically using chkconfig on Fedora

On Fedora, we can use the chkconfig to add Apache to the startup/shutdown sequence. chkconfig uses specially-formatted comments in the start/stop script to determine when a service is started: at which runlevels, and where in the sequence of starting/stopping services.

1. Make a symlink from the Apache control script to Fedora's init script directory:
   ```
   ln -s /opt/apache/bin/apachectl /etc/rc.d/init.d/apache
   ```
2. Add these extra lines to the top of /opt/apache/bin/apachectl:

   ```
   #
   # apache        Control script for the Apache HTTP Server
   #
   # chkconfig: 345 85 15
   # description: Apache web server
   ```

   The chkconfig line specifies:
   <run_levels> <start_priority> <stop_priority>
3. Add Apache to the services managed by chkconfig:
   ```
   chkconfig apache on
   ```
4. Confirm the configuration:
   ```
   chkconfig --list apache
   ```

   You should see something like this:
   ```
   apache   0:off   1:off   2:off   3:on    4:on   5:on   6:off
   ```
5. Once we have a script in /etc/rc.d/init.d, we can use a shortcut to start/stop services manually:

   ```
   service apache start
   service apache stop
   service apache restart
   service apache graceful
   ```

   etc.

# *General server limits*

There are a range of directives which govern the generic operating capacity of the server: for example, the maximum length of time to spend waiting for a client, the maximum number of client connections allowed, whether to use KeepAlive connections, and so on. The most important ones are:

```
# time to wait for slow clients; default is 300,
# but setting this lower improves resilience
```

```
# against DOS attacks
TimeOut 60
# keep-alive allows multiple HTTP requests to be
# served over a single TCP request;
# the client needs to explicitly mark itself
# as being capable of handling this type of request
# in a request header for Apache to serve the request this way
KeepAlive On
# the max. number of requests to serve over a single
# TCP connection; default is 100, but the
# Apache manual recommends setting it higher
MaxKeepAliveRequests 200
# length of time to keep a connection open while
# waiting for the next request in a keep-alive
# sequence; default is 5; lower it on heavily-loaded
# servers to prevent Apache leaving
# connections idling while they wait for clients
KeepAliveTimeout 15
# maximum size of request body (0 = no limit)
LimitRequestBody 0
# number of header fields allowed in a request
LimitRequestFields 100
# how long header fields can be (in bytes)
LimitRequestFieldsize 8190
# how long the initial line of a request can be
LimitRequestLine 8190
```

## *MPM settings*

We also need some directives to control the activity of the MPM. For the prefork MPM (which we're using) we can specify the following:

```
# number of spare servers to keep running to
# handle potential incoming requests
MinSpareServers 5
# max. number of servers to leave idling
MaxSpareServers 10
```

```
# number of servers to start when Apache boots
StartServers 5
# maximum number of clients to serve simultaneously
MaxClients 150
# maximum requests to handle by any one server instance before it
is restarted;
# default is 10000 (unlimited), but setting it lower will help in
cases where
# Apache modules are memory-leaking, as a single process will be
unable
# to consume too much memory; for keep-alive sessions, this
represents the number of clients
# per child, rather than requests per child
MaxRequestsPerChild 1000
```

If we're using the worker MPM, it has a different set of configuration directives. See the Apache manual for more details, or the sample config. file in /opt/apache/conf/extra/httpd-mpm.conf.

Try changing these values, and use ps to see the processes which Apache starts.

# *File layout*

We've already started making decisions about how our Apache server will be laid out. Let's consolidate this now.

Suggested directory layout:

- Binaries and supporting files: /opt/apache
- Default web site: /var/www/htdocs (we can eventually remove this if we don't want to use it)
- Default cgi-bin: /var/www/cgi-bin (ditto as for the htdocs directory)
- Log files: /var/log/apache (e.g. access_log, error_log)
- Process file: /var/run/apache

We're going to use a separate layout for each virtual host (to be covered later).

Although we have an apache user and group, they shouldn't own the Apache executables: these must start as root to run on port 80; so if someone cracked the apache user account and replaced the httpd binary with a Trojan'ed version, the next time httpd runs it would run the Trojan code as root.

## Summary of filesystem layout

| Path | User:group ownership | Directory permissions | File Permissions |
|---|---|---|---|
| /opt/apache | root:root | 755 | 644 |
| /opt/apache/bin | root:root | u+x | - |
| /opt/apache/build/*.sh | root:root | - | u+x |

| Path | User:group ownership | Directory permissions | File Permissions |
|---|---|---|---|
| | | | |
| /opt/apache/conf | root:root | 700 | - |
| /var/log/apache | root:root | 700 | - |
| /var/run/apache | root:root | 700 | - |
| /var/www/htdocs | root:root | 755 | - |
| /var/www/cgi-bin | root:root | 755 | - |

Here are the commands to implement these settings:

```
chown -R root:root /opt/apache
find /opt/apache -type d | xargs chmod 755
find /opt/apache -type f | xargs chmod 644
find /opt/apache/bin -type f | xargs chmod u+x
chmod u+x /opt/apache/build/*.sh
chmod 700 /opt/apache/conf
chown root:root /var/log/apache
chmod 700 /var/log/apache
chown root:root /var/run/apache
chmod 700 /var/run/apache
chown root:root /var/www/htdocs
chmod 755 /var/www/htdocs
chown root:root /var/www/cgi-bin
chmod 755 /var/www/cgi-bin
```

# *Logging*

[To enable logging, the log_config module has to be loaded first.]

Apache has two separate logs:

1. **Error log**
   The first stop for diagnosing errors with the server; by default will also contain CGI error output. Example message:

   ```
   [Wed Oct 11 14:32:52 2000] [error] [client 127.0.0.1] client
   denied by server configuration: /opt/apache/htdocs/test
   ```

   Date and time of the message
   Type of message
   IP address of client which triggered the error

Error message

The level of logging is set in Apache config. using the *LogLevel* directive. The possible settings are (in order of decreasing significance):

`emerg` Emergencies - system is unusable. "Child cannot open lock file. Exiting"
`alert` Action must be taken immediately. "getpwuid: couldn't determine user name from uid"
`crit` Critical Conditions. "socket: Failed to get a socket, exiting child"
`error` Error conditions. "Premature end of script headers"
`warn` Warning conditions. "child process 1234 did not exit, sending another SIGHUP"
`notice` Normal but significant condition. "httpd: caught SIGBUS, attempting to dump core in ..."
`info` Informational. "Server seems busy, (you may need to increase StartServers, or Min/MaxSpareServers)..."
`debug` Debug-level messages "Opening config file ..."

Setting the LogLevel tells Apache to log all messages of that severity or higher. Setting the LogLevel to crit, for example, will report emerg, alert and crit messages. The standard setting is **error**.

The log is written to the file specified by the *ErrorLog* directive, which specifies the path for the log file, e.g. `ErrorLog /var/log/apache/error_log`

2. **Access log**
   This logs requests made to the server. It is set up by defining two directives:

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
\"%{User-Agent}i\"" combined
CustomLog /var/log/apache/access_log combined
```

Here I am using a standard log format commonly known as "combined". Note that you can reference any request header using the *%{Header}i* syntax. You can also record response headers with *%{Header}o*.

%>s is the status sent in the response (e.g. 200, 404, 302). If you specify %>s, the final status is recorded; if you specify %<s, the initial status message sent to the request is recorded.

## Adding logging configuration

Putting this together for our setting gives us the following extra lines for httpd.conf:

```
# load shared modules
LoadModule log_config_module modules/mod_log_config.so

# error log
LogLevel info
ErrorLog "/var/log/apache/error_log"
```

```
<IfModule log_config_module>
     # access log
     LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
\"%{User-Agent}i\"" combined
     CustomLog "/var/log/apache/access_log" combined
</IfModule>
```

Note I sneaked in a directive to load a shared module here (mod_log_config.so). This is necessary before we can start using the directives which that module makes available in our config..

I also put the directives which depend on this module inside a conditional <IfModule> directive. This means that if we decide to turn off this module at some point, the directives relating to it are ignored. This makes the config. file more stable, and also makes it easier to track dependencies between modules and directives.

## Log rotation using rotatelogs and pipes

Apache comes with a utility for rotating logs called rotatelogs. You can specify that this be used in the CustomLog directive by specifying a pipe ( | ) for the CustomLog:

```
CustomLog "|/opt/apache/bin/rotatelogs -l
/var/log/apache/access_log-%Y-%m-%d 86400" common
```

(This command rotates the access log every 24 hours, and calls the old logfile access_log suffixed with the full year, month and day; 86400 = 24 hours = 60 * 60 * 24 seconds; the -l option forces the server to use local time for the logs rather than GMT)

It is also possible to rotate the logs based on size (replace the time specification with a file size, e.g. 5M)

There is another log rotation script called **cronolog** (http://cronolog.org/), which offers finer-grained control over logging, but which can be used in the same way as rotatelogs (i.e. via a pipe).

## Log rotation using logrotate

**logrotate** is another solution available with most Linux distributions. It works externally to the programs it is rotating for: you don't configure it inside httpd.conf, but configure logrotate itself instead, telling it which logs to rotate. logrotate can be used to rotate logs for any application, and runs as a daemon. Here's a sample configuration script for rotating our Apache logs (adapted from Ubuntu's logrotate configuration for Apache):

```
/var/log/apache/*_log {
  # rotate on a daily basis
  daily
  # don't return an error if there are no *_log files
  missingok
  # keep 30 copies of logs
  rotate 30
```

```
  # compress rotated logs
  compress
  # wait for another rotation before compressing logs
  delaycompress
  # create new log files with mode 600, owner root, and group root
  create 600 root root
  sharedscripts
  # script to run after rotating logs
  postrotate
    if [ -f /var/run/apache/httpd.pid ]; then
      /opt/apache/bin/apachectl graceful > /dev/null
    fi
  endscript
}
```

Here's a good reference for creating your own logrotation scripts, and what the directives mean:

http://www-uxsup.csx.cam.ac.uk/~jw35/courses/apache/html/x2167.html

The location to put the configuration file into depends on how the logrotate daemon is configured on the machine; in the case of Fedora, the above configuration script would be placed in:

/etc/logrotate.d/apache

You can test your logrotate script manually using:

logrotate -f /etc/logrotate.d/apache

## Custom log rotation scripts

It's pretty easy to write your own log rotation script which works offline. This is more efficient than using piped logs, as it only requires a short-lived process which runs occasionally to archive the log files (unlike rotatelogs, which runs continuously with Apache). However, it may be a less sustainable choice than a dedicated application like logrotate (see earlier), as you have to maintain the script yourself, though it should be easier to setup.

Here's a sample script we could use with cron (as the root user) to rotate our logs on a daily basis:

```
#!/usr/bin/python
import time
from subprocess import call
from os import rename
suffix = '.' + time.strftime('%Y-%m-%d')
access_log = '/var/log/apache/access_log'
archived_access_log = access_log + suffix
```

```
error_log = '/var/log/apache/error_log'
archived_error_log = error_log + suffix
rename(access_log, archived_access_log)
rename(error_log, archived_error_log)
# do a graceful restart
call(['/opt/apache/bin/apachectl', 'graceful'])
```

While saving some CPU cycles, this approach also has the advantage of keeping log file names simple (just access_log and error_log), as logrotate does. This makes configuration easier later on (e.g. if we want multiple virtual hosts to write to the same access_log, we can just specify the filename access_log).

The old log files are renamed by appending a date suffix onto the end of the original file name. You could refine this by removing really old logs, or zipping the archived logs.

[NB there appears to be a bug with the graceful restart command for Apache 2.2 (it is recorded on the Apache bug tracker), which causes an error to appear in the logs when running the above script. However, this appears to have no effect on the server's operation.]

# *Configuring file serving*

## Safe defaults for serving directories

By default, Apache will serve any file it can access. This could be problematic if a mis-configuration made it possible for Apache to serve critical system files. We can set the default to deny access to the whole filesystem by default:

```
<Directory />
     Order Deny,Allow
     Deny from all
</Directory>
```

The <Directory> directive allows you to group a set of options which apply to a specified directory in the filesystem (and all its sub-directories). In our case, we are applying it to / (the root of the whole filesystem).

The Order directive is part of the host based authentication module (mod_authz_host). It specifies the order in which Deny and Allow directives are applied. In this case, Deny directives are applied first, then Allow directives. Access is allowed by default. Any client which does not match a `Deny` directive or does match an `Allow` directive will be allowed access to the directory.

The Deny directive specifies that all hosts are denied access. It is possible to restrict access using IP addresses, partial IP addresses, network/netmask pairs, or network/nnn CIDR specification, e.g.

Allow from 82.68.194.150

Allow from 10.1

Allow from 10.1.0.0/255.255.0.0

Allow from 10.1.0.0/16

You can also control access by environment variable using:

Allow from env=access_granted

Using setenvif, you could set environment variables based on arbitrary features of the request (e.g. particular user agents, referer, non-standard headers), which could then be used to grant/deny access.

We now need to allow access to the default website directory so we *can* serve files from it:

```
<Directory /var/www/htdocs>
     Order Allow,Deny
     Allow from all
</Directory>
```

We need to add these directives, plus the LoadModule statement to pull in the module which controls authentication, to httpd.conf:

```
...
LoadModule authz_host_module modules/mod_authz_host.so
...
<Directory />
     Order Deny,Allow
     Deny from all
</Directory>
<Directory /var/www/htdocs>
     Order Allow,Deny
     Allow from all
</Directory>
```

We now have enough in place to test whether we can serve files. All we need to do now is:

1. Change to the root user
2. Create a file in /var/www/htdocs called index.html (any content)
3. Restart Apache
4. Go to http://localhost/index.html: you should see your content

## Options on directories

The Options directive covers file access configuration for individual <Directory> directives. It

allows you to govern features like execution of files, following symlinks, and showing indexes of files in a directory. Here are the options available:

- **ExecCGI:** CGI scripts can be executed in the directory
- **FollowSymLinks:** symlinks in the directory can be followed to their target, even if outside the webserver's document tree (NB this one is needed for mod_rewrite, which is very important for URL rewriting and used by many applications for Search Engine Optimisation of URLs)
- **SymLinksIfOwnerMatch:** only follow symlinks if the owner of the link is the same as the owner of the file pointed to
- **Includes:** allows server-side includes
- **IncludeNOEXEC:** allows server-side includes, but prevents the exec command being used in SSIs
- **Indexes:** when on, the server will generate an index of files in a directory if no default resource (liked index.html) is specified
- **MultiViews:** allows content negotiation (i.e. serve files based on user's language preference)
- **All:** all of the above are enabled except MultiViews
- **None:** none of the above are enabled

To enable or disable an option, use this syntax:

```
  Options +FollowSymLinks
  Options -Indexes +ExecCGI
```

To see why symlinks are important, try this:

1. ln -s /etc/passwd /var/www/htdocs/passwd
2. Go to http://localhost/passwd

This isn't too dangerous here, as we need to be root to create the symlink (no one else can write into /var/www/htdocs/). But if we had configured the site to allow access by non-root users who can read /etc/passwd, they would be able to do the same thing for their own website.

To prevent this kind of thing, we should set `Options None` for the directive which covers the root of the filesystem:

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

This now becomes the default setting for any directories below the root of the filesystem, including /var/www/htdocs.

## AllowOverride: overriding server configuration in a directory

This directive governs whether parts of the server configuration can be overridden using files inside the webserver document tree. For example, we may allow users to specify their own authorisation directives in these files, to govern which hosts, users, and or groups can access their directories. Configuration is overridden in **.htaccess** files.

The following options specify which parts of the configuration can be overridden in .htaccess files:

| **Option** | **Controls overriding of this type of directive...** |
|---|---|
| AuthConfig | Authorisation, e.g. Require, AuthUserFile, AuthType |
| FileInfo | Document type, e.g. Header, ErrorDocument, RewriteBase |
| Indexes | Directory indexing, e.g. IndexOptions, DirectoryIndex, DefaultIcons |
| Limit | Host access, e.g. Allow, Deny, Order |
| Option | The Options directive |
| All | Any directive which can be overridden in .htaccess files can be overridden in this directory (i.e. all of the above) |
| None | None of the above; .htaccess files are ignored |

The Options override is probably the most confusing: if AllowOverride Options is specified, then the default Options setting for the directory can be overridden by a .htaccess file in that directory!

You can specify which directive types can be overridden like so:

```
AllowOverride AuthConfig Limit
```

In our case, for the root directory, we don't want to allow anything to be overridden:

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

## Hiding important files

By default, Apache will serve any file requested which is within a visible directory. This includes .htaccess files (discussed above) which may contain important configuration information; plus it could contain backup files (commonly ending with .bak or starting with ~, depending on the editor which produced them).

Try adding a .htaccess file to /var/www/htdocs, then fetch it in your web browser. It should work OK, which isn't what we want.

We can globally turn off access to these files like this by putting a FilesMatch directive at the top level directive of our httpd.conf file:

```
<FilesMatch "(^\.ht|~$|\.bak$)">
  Order Deny,Allow
  Deny from All
</FilesMatch>
```

This directive can also be applied to individual virtual hosts or directories, and can be set in .htaccess files if AllowOverride is set to All for that directory.

Now try getting your .htaccess file. It should be protected.

There is also a DirectoryMatch directive, which can be used to prevent serving of directories whose name matches a specified regular expression.

## Setting the default home page

One useful thing we can do immediately is define the default document to serve when the root of a directory is requested, e.g. http://localhost/. We do this with the DirectoryIndex directive, which needs mod_dir to be loaded:

```
LoadModule dir_module modules/mod_dir.so
<IfModule dir_module>
  DirectoryIndex index.html
</IfModule>
```

(Test it at http://localhost/)

When we add other types of file (e.g. PHP scripts), we can add these onto the DirectoryIndex to make them available as the default index page.

## Setting the right MIME types

When you fetch index.html, you'll probably notice that it turns up as plain text. If you check the response headers when you fetch index.html, you'll notice the resource is delivered with the MIME type text/plain. However, we would expect any .html file to be treated as text/html. This is because we haven't configured MIME handling. This facility is provided by mod_mime, and activated like this:

```
LoadModule mime_module modules/mod_mime.so

### mime types
```

```
DefaultType text/plain
<IfModule mime_module>
  # location of the MIME types configuration file
  TypesConfig conf/mime.types
</IfModule>
```

The mime.types file maps file suffixes (.html, .php etc.) to MIME types (a MIME type just describes the kind of content a file contains, and is used by the client to determine how to handle the file, e.g. display in the browser, download, display in a helper application). Note that the TypesConfig directive is implicit and doesn't have to be specified as we have here, and it will still work. But it's worth being explicit, again to remind us of the dependency between the module and the mime.types file in the conf directory.

There is another MIME module called mod_mime_magic, which uses hints in the file to determine its MIME type, as well as the filename suffix. This could be helpful in cases where you have many unusual and esoteric file types, or have files without suffixes or incorrect suffixes.

It is also possible to add your own custom MIME types on top of the default ones using mod_mime.

### Compressing content sent to the client

This is a useful option, and one which can reduce network bandwidth usage. It enables Apache to compress content sent to clients that are able to handle such compressed content (i.e. most modern browsers).

1. Enable mod_deflate:

   ```
   LoadModule deflate_module modules/mod_deflate.so
   ```

2. Configure compression for common content types:

   ```
   AddOutputFilterByType DEFLATE text/html text/plain text/xml
   ```

It is possible to compress other types of content, but configuration is more complex and requires browser sniffing (see http://httpd.apache.org/docs/2.2/mod/mod_deflate.html). This configuration is straightforward and will work with all browsers.

NB Apache will only send compressed content to clients whose requests include the following header:

```
Accept-Encoding: gzip,deflate
```

We can test this by requesting our index.html file, then checking the response headers which come back from Apache. They should include:

```
Content-Encoding: gzip
```

## *Hiding the server's identity*

The response we get back when we request a resource on the server gives away some information about the server. Namely, the response contains a Server header which looks like this:

```
Server: Apache/2.2.2 (Unix)
```

We can see this using the LiveHTTPHeaders in Firefox.

An attacker could use this information to potentially determine vulnerabilities in the server, based on the server type, version, and underlying operating system. There are two simple things we can do to hide this information in httpd.conf:

```
# this line controls whether Apache adds information about
# itself to the end of server-generated documents
# (e.g. directory index pages, error messages);
# Off is the default, but let's make it explicit
ServerSignature Off
# the tokens displayed in response headers;
# this sets it to just show the server name (Apache);
# this can only be set at the server level (not per host)
ServerTokens ProductOnly
```

If you are really paranoid, and want to disguise the fact you are using Apache at all, you can change the Server header in the response to whatever you like using the mod_security module (we're not going to bother):

```
ServerTokens Full
SecServerSignature "Elliot's Miraculous Web Server"
```

You can get mod_security from:

http://www.modsecurity.org/projects/modsecurity/apache/index.html

It's very easy to install (using the instructions for compiling new Apache shared modules - see earlier).

However, there are still certain aspects of the behaviour of the server's networking stack and the way it formats responses which can enable the server's real identity to be discovered.

## *chrooting*

Chroot'ing Apache is another way to add more security, by constricting Apache to running in a specific directory. No files outside the chroot directory are accessible to Apache once running.

The traditional method for chroot'ing Apache is complex; however, mod_chroot is an easier way to chroot Apache which keep things simple: http://core.segfault.pl/~hobbit/mod_chroot/

# CGI

"The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers, such as HTTP or Web servers. A plain HTML document that the Web daemon **retrieves** is **static**, which means it exists in a constant state: a text file that doesn't change. A CGI program, on the other hand, is **executed** in real-time, so that it can output **dynamic** information." (from **http://hoohoo.ncsa.uiuc.edu/cgi/intro.html)**

## *Apache and CGI*

CGI scripts run as processes external to Apache, and run as the effective Apache user. Each time a CGI script is requested by a client, a new process is fired up to handle it. (This is fairly inefficient, and several solutions exist to alleviate this, as described later. It also means that a poorly-written CGI script can hog memory and CPU cycles: again, the solutions described later go some way to helping with this.)

Common practice is to put CGI scripts into a dedicated directory. This is the most secure way of hosting scripts, but the least flexible from the user's perspective.

1. Create a separate cgi-bin folder in /var/www/cgi-bin
2. chmod 755 /var/www/cgi-bin
3. Setup CGI config. for that directory in /opt/apache/conf/httpd.conf:

   ```
   LoadModule cgi_module modules/mod_cgi.so

   <Directory /var/www/cgi-bin>
     Order Allow,Deny
     Allow from all
   </Directory>
   ```

4. We need to load mod_alias so we can alias a directory which holds CGI scripts:

   ```
   LoadModule alias_module modules/mod_alias.so
   ```

5. Create an alias for the cgi-bin directory:

   ```
   ScriptAlias /cgi-bin/ /var/www/cgi-bin/
   ```

   This directive means that any file put into the /var/www/cgi-bin/ directory is treated as a CGI script; also that any URL of this form:

   http://localhost/cgi-bin/*filename*

   is mapped onto a script called *filename* in the /var/www/cgi-bin/ directory.

6. Create a test CGI script (I'm using Python) in the cgi-bin directory:

   ```
   #!/usr/bin/python
   print "Content-Type: text/plain"
   print "\n"
   print "Hello world"
   ```

7. Make the script executable:

   chmod 755 hello.py

8. Try accessing it at: http://localhost/cgi-bin/hello.py

It is safe to use ScriptAlias where we are setting up a directory to execute CGI scripts which is outside the document root for the server (i.e. the directory is not available via any means other than through the ScriptAlias). However, where we want to allow CGI execution inside a directory under the document root, it is better to use the <Directory> directive instead.

For example, if we wanted to allow Python CGI scripts under /var/www/htdocs, we could enable them like this:

```
<Directory /var/www/htdocs>
  Order Allow,Deny
  Allow from all
  Options ExecCGI
  AddHandler cgi-script .py
</Directory>
```

# Improving security with suEXEC and FastCGI

suEXEC and FastCGI are two alternative mechanisms for adding extra security and stability to CGI script execution.

- **suEXEC** only really makes sense in a shared hosting environment with virtual hosts. It allows execution of CGI scripts as a user different from the Apache effective user. To use it, Apache must be compiled with suEXEC support (it isn't compiled in by default). Once in place, a virtual host can be configured to run any CGI scripts under a different user and group, as specified by the SuexecUserGroup directory (only allowed inside a VirtualHost directive). suEXEC puts a stringent series of checks in place every time a CGI script is requested, such as checking whether the suEXEC user exists, whether they have permissions to execute the script, permissions on the directory containing the script, ownership of the script, and so on.
- **FastCGI** creates a separate process for CGI scripts, and allows requests for specified resources to be routed to that process. Because the FastCGI process is persistent across requests (so multiple requests for a CGI script can be handled by a single process), it is more efficient than standard CGI (close to Apache module speeds). In addition, FastCGI can also be configured to run under suEXEC, so the external FastCGI process runs as a user different from the Apache effective user.

  The only downside to FastCGI is that it is not in active development, and is widely considered "abandoned" (for example, the README has not been updated for 2 years, and the current version is not compatible with Apache 2.2 without manual patching of the source).

# SSL

SSL is a protocol for communicating securely between a client and a server. Originally developed to secure HTTP communications, it has been extended to cover SMTP, IMAP and other protocols.

SSL is based around Public Key Cryptography. In this type of encryption, there isn't a single key (as there is in symmetrical encryption); there are two keys, one public and one private. Anything encrypted with the public key can be decrypted only if the private key is known; and anything encrypted with the private key can be decrypted only with the public key.

SSL works in two phases:

1. **Handshake**
   The server sends the client a digital certificate, which contains its public key. The certificate can either be signed by:
   1. The owner of the certificate (a.k.a. self-signed, shouldn't be trusted)
   2. A private certificate authority (e.g. an organisation might create a certificate for use on its intranet)
   3. A public certificate authority (i.e. an organisation which exists only to sign certificates and verify identities)

   A certificate received by a client may have been signed directly by one of the above, or by an intermediary between the certificate's owner and a certificate authority. The client verifies the certificate by following the signing chain of the certificate back to its root authority, and makes a decision to either trust or reject the certificate. Optionally, the server may require that the client send its own certificate before it will allow communication.

   If the certificate is trusted, the client and the server then negotiate the encryption protocol to use and a set of symmetrical keys. Symmetrical keys are faster than using public key encryption.
2. **Data exchange**
   The client and server exchange data using the agreed symmetrical key.

There is a good document explaining the ins and outs of SSL and Apache at:

http://www.modssl.org/docs/2.8/

## *Creating a self-signed certificate*

To generate a self-signed SSL certificate, you will need openssl installed.

Then follow these steps:

1. Generate the server's private key; we'll use a 1024-bit key using the RSA algorithm:

   ```
   cd /opt/apache/conf
   mkdir ssl
   cd ssl
   openssl genrsa -out server.key 1024
   ```
2. Generate a certificate-signing request:

   ```
   openssl req -new -key server.key -out server.csr
   ```

Fill in the required information. The important fields are:

```
Country Name (2 letter code) [GB]:GB
State or Province Name (full name) []:.
Locality Name (eg, city) [Newbury]:Birmingham
Organization Name (eg, company) [My Company Ltd]:mooch labs
Organizational Unit Name (eg, section) []:.
Common Name (eg, your name or your server's hostname)
[]:localhost
Email Address []:.
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:.
An optional company name []:.
```

The really important one is the Common Name: this must match the domain name which will serve the SSL site; otherwise connecting clients will get a prompt about a mismatch between the certificate's host name and the actual host name of the server.

Note that we left the password blank. If we don't do this, Apache will prompt you for the certificate password each time you start the server.

4. Create a self-signed certificate:

   openssl x509 -req -days 3650 -in server.csr -signkey server.key -out server.crt

5. rm server.csr
   (we don't need it any more)
6. You can view the certificate using this command:

   openssl x509 -text -in server.crt

# Configuring Apache to use SSL

We're going to put our SSL keys into a separate directory /opt/apache/conf/ssl, and the configuration in a separate file called /opt/apache/conf/ssl.conf.

1. Compile mod_ssl statically into the server, or load it as a shared module; we compiled the shared module earlier, so we load it with:

   ```
   LoadModule ssl_module modules/mod_ssl.so
   ```

2. We're also going to need the SetEnv module for some of the configuration we need to do later:

   ```
   LoadModule setenvif_module modules/mod_setenvif.so
   ```

3. Make sure the private and public keys are in the right directory:

ls /opt/apache/conf/ssl

You should see server.key and server.crt.

4. Set permissions on the directory:

   chown root.root /opt/apache/conf/ssl
   chmod 700 /opt/apache/conf/ssl

5. Set permissions on the certificate and the key:

   chmod 600 /opt/apache/conf/ssl/server.*

6. Make a new file to hold Apache's SSL configuration:

   touch /opt/apache/conf/ssl.conf
   chmod 600 /opt/apache/conf/ssl.conf

7. Make a directory to store the SSL session cache (this improves performance as it caches session data and prevents unnecessary handshakes, e.g. if a single client creates multiple parallel connections to the server):

   mkdir /opt/apache/cache
   chown root.root /opt/apache/cache
   chmod 700 /opt/apache/cache

8. Put together a minimal SSL configuration in ssl.conf:

```
Listen 443
SSLCertificateFile conf/ssl/server.crt
SSLCertificateKeyFile conf/ssl/server.key
# switch off SSLv2 (which is flawed)
SSLProtocol All -SSLv2
# only support high-grade encryption
SSLCipherSuite ALL:!EXP:!NULL:!ADH:!LOW
# session cache: type:location(max_size)
SSLSessionCache shmcb:/opt/apache/cache/sslcache(51200)
SSLSessionCacheTimeout 300

# configuration to handle broken SSL implementation
# in IE
SetEnvIf User-Agent ".*MSIE.*" \
  nokeepalive ssl-unclean-shutdown \
  downgrade-1.0 force-response-1.0

# configure the default site to be available over SSL
# as well as standard HTTP
<VirtualHost localhost:443>
  SSLEngine on
  ServerName localhost:443
  DocumentRoot /var/www/htdocs
  CustomLog /var/log/apache/access_log combined
  ErrorLog /var/log/apache/error_log
</VirtualHost>
```

9. Pull the configuration file into the main httpd.conf file:

   ```
   Include /opt/apache/conf/ssl.conf
   ```

10. Test at:

    https://localhost/

    Note that you will be prompted to accept the certificate, as it is self-signed and cannot be traced back to a recognised certificate authority.

# Adding PHP

## Pre-installation

There are several choices to make:

1. **Which version: 4 or 5 or both?**
   PHP 5 is stable, and superior to version 4 in its support for object-oriented programming. It is also possible to run PHP 5 in version 4 compatibility mode, which should provide near-perfect support for PHP 4 scripts.

   An alternative is to install both, and select the version to use as follows:

   1. per-host (by setting an AddHandler directive for a whole virtual host which specifies the PHP version to use)
   2. per-directory (by setting an AddHandler directive inside a directory, either in a .htaccess file or in httpd.conf)
   3. per-file (by setting a handler for files with a specific file suffix, e.g. .php4, in httpd.conf or .htaccess)

   We are going to install PHP 5.
2. **Do you want web, command line, and/or GUI?**
   If you don't need command line or GUI support, leave them out when compiling PHP.
3. **Will it be used by untrusted users?**
   In situations where the server will only be used by trusted users, PHP can safely be run as a module. In this situation, PHP runs inside the main server process, under the Apache effective user. Where some untrusted users may be using the server to run PHP scripts, a safer setup is to use standard CGI, CGI with an execution wrapper like suEXEC, or PHP under FastCGI. This isolates the PHP process from Apache and is safer; it also means that Apache is potentially faster, as it isn't also running PHP, so static file delivery may be speeded up.

   We are going to install as a module, as this is the simplest approach, and good for most general purpose use.

## Preparation

You will need the following pieces of software to compile PHP on Ubuntu:

- flex
- bison
- autoconf
- MySQL
- MySQL-dev (libmysqlclient14-dev on Ubuntu)
- libjpeg-dev, libpng-dev, libxpm-dev, libwmf-dev, libungif, libfreetype6-dev etc. (to get support for different image formats and truetype fonts in GD)

# *Compiling PHP*

Download from php.net

Compare with the md5sum (as we did for Apache)

Unpack

Connect to the unpacked directory

To compile PHP, we need to reference a couple of graphics library files. On Ubuntu, this isn't a problem; but on Fedora (at least in version 5), the graphics libraries have non-standard names which cause compilation to fail. We can fix this by symlinking the real graphics libraries to correctly-named files like this:

```
ln -s /usr/lib/libjpeg.so.62 /usr/lib/libjpeg.so
ln -s /usr/lib/libXpm.so.4 /usr/lib/libXpm.so
```

Run the configure script like this:

```
./configure --prefix=/opt/apache/php --with-apxs2=/opt/apache/bin/apxs --with-config-file-path=/opt/apache/conf --enable-memory-limit --with-pear=/opt/apache/php/pear --without-pgsql --with-mysql=shared --with-mysqli=shared --with-pdo-mysql=shared --with-gd=shared --with-zlib=shared --with-freetype-dir=/usr/lib --with-xpm-dir=/usr/lib --with-jpeg-dir=/usr/lib --with-gettext=/usr/lib
```

The options I've used here specify the following:

--prefix = where to install

--with-apxs2 = location of the apxs binary (for installing the PHP module into Apache)

--with-config-file-path = where the php.ini file will be

--enable-memory-limit = compile with memory limit support

--with-pear = install pear (packaging mechanism for PHP extensions)

--without-pgsql = disable support for PostgreSQL

--with-EXTENSION=shared = enable the following extensions as shared

- mysql = include support for MySQL
- mysqli = improved MySQL extension
- pdo-mysql = enable PDO support for MySQL (PDO is a new database interface in PHP 5)
- zlib = enable support for the zlib extension (stream compression support)
- gd = enable PHP GD support (for image manipulation and creation)

--with-freetype-dir, --with-xpm-dir, --with-jpeg-dir = path to Freetype/XPM/JPEG handling libraries (NB compiling against Freetype is the easiest way to enable PHP font-rendering functions within GD)

--with-gettext = location of the GNU gettext libraries; useful for internationalisation

Note that there is a default set of extensions installed with PHP which is fairly sane, so we will leave them as is. If you want to turn any of them off, use:

--disable-EXTENSION

OR

--without-EXTENSION

(use `./configure --help` to work out which you'll need for a given extension)

Then run these commands to compile and install:

make

make install

Next we need to copy the recommended PHP config. file to the location where we told our compiled PHP it would be:

cp php.ini-recommended /opt/apache/conf/php.ini
chown root:root /opt/apache/conf/php.ini
chmod 600 /opt/apache/conf/php.ini

When we ran `make install`, it added this line to /opt/apache/conf/httpd.conf:

```
LoadModule php5_module modules/libphp5.so
```

(If you recompile PHP and do `make install`, it may add another line like this to httpd.conf, which will break Apache. You can fix it by just removing the repeated line.)

Tell Apache which files to treat as PHP scripts:

```
AddHandler application/x-httpd-php .php
```

And to treat index.php as a possible default home page when a website root is requested:

```
DirectoryIndex index.html index.php
```

To test our installation:

1. cd /var/www/htdocs
2. create a new file called info.php with this content:

   ```
   <?php
   phpinfo();
   ?>
   ```

3. Test at http://localhost/info.php

   You should see a screen with information about your PHP settings, loaded modules, etc.

### A note on SELinux

If you follow these instructions on a default Fedora install, you may find that you are unable to restart Apache once you've installed PHP, and get an error something like:

> *Cannot restore segment prot after reloc: Permission denied*

This can be caused if you have SELinux enabled, which is the default on Fedora. (On Ubuntu, it isn't a problem (by default).)

The easiest way round this (I'm not going into the intricacies of SELinux policies here!) is to disable SELinux in /etc/selinux/config, by setting:

```
SELINUX=disabled
```

## Removing PHP

If for any reason you want to junk your installation, you can remove all traces of PHP like this:

1. Stop Apache
2. rm /opt/apache/php
3. rm /opt/apache/modules/libphp5.so
4. Delete or comment out this line:

   ```
   LoadModule php5_module        modules/libphp5.so
   ```

   in /opt/apache/conf/httpd.conf
5. The lines which set up the PHP handler and specify index.php as a DirectoryIndex can be left in, as they won't affect Apache's operation; remove them if you really want to get rid of PHP altogether
6. You can leave /opt/apache/conf/php.ini where it is: it won't affect Apache's operation.
7. Start Apache

NB this is one of the advantages of not spreading PHP and PEAR across the filesystem: it makes it easy to completely strip it out of the server. I found this really useful when experimenting with different configure switches.

## Extensions

Compiling extensions as static or shared is similar to Apache.

By default, the php.ini file doesn't load extensions; you have to tell it where they are and which ones to load.

Edit php.ini:

1. Set the directory containing PHP extension .so files:

   ```
   extension_dir = /opt/apache/php/lib/php/extensions/no-debug-non-zts-20050922
   ```
2. Add one directive for each extension:

   ```
   extension=mysql.so
   ```

   ```
   extension=mysqli.so
   ```

```
extension=pdo_mysql.so
extension=gd.so
extension=zlib.so
```

If you want to check the extensions which have been compiled in as shared, have a look in the extension_dir (defined above). You should see a .so file for each shared module.

You can also see a list of all extensions by doing:

/opt/apache/php/bin/php -m

though this doesn't discriminate between shared and static extensions.

# *Recompiling PHP*

## 1. Adding a new extension

We can compile new extensions into our PHP installation using the phpize tool. This is similar to apxs, but intended for installing PHP extensions. We'll install the mbstring extension this way:

1. Go to the PHP source tree
2. cd ext/mbstring
3. /opt/apache/php/bin/phpize
   This prepares the source in the current directory for compilation as a PHP extension
4. ./configure --with-php-config=/opt/apache/php/bin/php-config
5. make
6. make install
7. Edit /opt/apache/conf/php.ini and add this line:

   ```
   extension=mbstring.so
   ```

8. Check the extension is loaded using:

   /opt/apache/php/bin/php -m

   or by using phpinfo().

## 2. Recompiling the PHP binary

If we re-run ./configure at the top of the source tree with extra options, the PHP binary will be reconfigured. As far as I can tell, it's best to do a "make clean" to clean the previously-compiled version completely out of the build tree (NB this doesn't affect the installed PHP, just the build tree). We can then do the standard make/make install to update the PHP binary inside our Apache installation.

# *Configuring PHP*

We've already checked our PHP configuration using the phpinfo() command.

The config. file consists of a bunch of directives; if the directive is commented with a semi-colon,

the default value shown is set.

Now we are going to have a look at the configuration file and systematically cut it down and tighten it up.

1. Make a copy of the file (before we start butchering it).
2. Remove the big blocks of comments. This just makes the config. file a bit easier to read.
3. Delete any sections in the config. file which don't apply to our setup (i.e. for configuring extensions we're not using). Start at the end of the file and remove any sections headed [] which aren't required.
4. Add pear to the include path:

   ```
   include_path = ".:/php/includes:/opt/apache/php/pear"
   ```

   This ensures that if we install any PEAR extensions, they are available to our PHP scripts.
5. Starting from the top and working down:
   i. **safe_mode:** When safe_mode is on, PHP does a check when a script calls a function which tries to access a file on the filesystem. If the owner of the script and the owner of the file are different, PHP does not allow the operation. (NB this can be relaxed using the safe_mode_gid directive.)
   ii. **expose_php:** turn it to "Off" if you don't want PHP to add itself to the Apache response headers.
   iii. **memory_limit:** 8M is quite low, and may cause problems with certain scripts; a setting of 64M is more realistic.
   iv. **display_errors:** Leave this off on a production server and log errors to a file instead. You can turn it on in individual scripts if you need it with:

   ```
   ini_set("display_errors", 1);
   ```

   You should also make sure **display_startup_errors** is set to Off.
   v. **error_log:** log errors into a file, rather than displaying them in the response:

   ```
   error_log = "/var/log/php/php_log"
   ```

   NB log_errors must be set to On for this to work.
   vi. **register_globals:** set to Off. Do not turn it on: it is very dangerous and can open vulnerabilities in poorly-written scripts.
   vii. **allow_url_fopen:** set to Off. If On, this allows PHP scripts to open files on remote servers via ftp or http.
   viii. **magic_quotes_gpc:** set this to Off. It is confusing if it's turned on, as it automatically escapes quotes in POST data.
   ix. **file_uploads:** turn on if you want to globally allow file uploads via PHP scripts.
   x. **enable_dl:** turn this Off; if On, it allows users to load their own extensions from within a PHP script.
   xi. **sendmail_path:** set the path to the sendmail binary if it is in a non-standard location,

or not on the apache user's path

xii. **session.save_path:** the path to the directory into which session data is saved; set it to /var/www/sessions

xiii. **session.referer_check:** set to the domain name for the Apache server; this ensures that session cookies are only accepted if the client's referer contains this string; in our case, we can set it to localhost.

As PHP runs as the apache user, and we have tightened access to /var/log/apache, we will setup a separate log directory for PHP. This directory will be writeable by the apache user (/var/log/apache isn't, for security reasons, and rather than make /var/log/apache writeable, it's better to put PHP logs into a different, less-secure directory):

mkdir /var/log/php

chown apache.apache /var/log/php

chmod 700 /var/log/php

(We could also apply log rotation to these logs, as we did for the Apache logs.)

We also need a separate directory to save session data:

mkdir /var/www/sessions

chown apache.apache /var/www/sessions

chmod 700 /var/www/sessions

## *Testing PHP + MySQL*

(I'm assuming you have a MySQL setup on your machine. I'm not going to explain how to do that :).)

First we need a database, a table, and some data:

1. Start the mysql command line client in a terminal
2. At the mysql prompt:

   use test;

   create table people (id INT AUTO_INCREMENT, name VARCHAR(255), PRIMARY KEY(id));

   insert into people values(1, 'Elliot Smith');

   insert into people values(2, 'Mickey Mouse');

   exit;
3. Write a PHP script to access our MySQL database (not secure - root has no password in my example!):

```
<?php
mysql_connect('localhost', 'root');
mysql_select_db('test');
$result = mysql_query('SELECT * FROM people');
```

```
while($row = mysql_fetch_assoc($result)) {
  echo $row['name'] . '<br/>';
}
?>
```

And a short script using PDO's MySQL functionality:

```
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', 'root');
foreach ($dbh->query('SELECT * FROM people') as $row) {
  echo $row['name'] . '<br/>';
}
$dbh = null;
?>
```

## Testing PHP's GD extension

We can test the GD PHP extension with a short script. It's worth doing this, as GD relies on several other installed libraries, and it's best to check they are being referenced correctly.

Create a new file in /var/www/htdocs/gd_test.php with this content:

```
<?php
$im = imagecreatetruecolor(400, 100);
$black = imagecolorallocate($im, 0, 0, 0);
$white = imagecolorallocate($im, 255, 255, 255);
$font = '/var/lib/defoma/x-ttcidfont-
conf.d/dirs/TrueType/Arial_Black.ttf';
imagefilledrectangle($im, 0, 0, 400, 100, $white);
imagettftext($im, 30,  0, 10, 40 , $black, $font, 'Hello World!');

header('Content-Type: image/png');
imagepng($im);
?>
```

You may need a different font path: use

locate ttf

or

find / -name *.ttf

to find the TrueType fonts on your system.

On Fedora, you could use:

/usr/share/fonts/bitstream-vera/Vera.ttf

for example.

Test by browsing to http://localhost/gd_test.php

# .htaccess files

These files can be used to set local configuration for a directory (and its subdirectories). They are commonly used to specify authentication and authorisation setup, but can also be used to set custom handlers, rewrite rules, PHP configuration, and so on (in fact, you can set any directives enabled for the directory, as specified by AllowOverride).

Note that any configuration you can do in a .htaccess file can also be done inside the main Apache configuration files. If you have control over the main config. files, use them instead of doing configuration inside .htaccess files, as it means your config. will be centralised and easier to manage.

## *Setting up authentication by username and password*

1. Switch to the root user
2. Allow configuration for the document root directory to be overridden in .htaccess files by modifying httpd.conf:

   ```
   <Directory /var/www/htdocs>
     AllowOverride FileInfo AuthConfig Limit
   </Directory>
   ```

3. Create the directory we want to secure:

   mkdir /var/www/htdocs/secure

4. Create an index.php file inside the secure directory.
5. We need to load the modules required to do user and group authentication and authorisation:

   ```
   LoadModule authn_file_module modules/mod_authn_file.so
   LoadModule auth_basic_module modules/mod_auth_basic.so
   LoadModule authz_user_module modules/mod_authz_user.so
   LoadModule authz_groupfile_module modules/mod_authz_groupfile.so
   ```

6. Create a data directory which will contain the configuration files for authentication:

   mkdir /opt/apache/data
   chown root:root /opt/apache/data
   chmod 711 /opt/apache/data

7. Create the file with the user data using the htpasswd program:

   /opt/apache/bin/htpasswd -c /opt/apache/data/passwords elliot

   The -c switch tells the command where to create the passwords file; elliot is the user we are creating. You will be prompted to enter a password then confirm it.

8. Create a .htaccess file in /var/www/htdocs/secure/.htaccess to protect the secure directory:

   ```
   AuthType Basic
   ```

```
AuthName "Secure area"
AuthUserFile /opt/apache/data/passwords
Require valid-user
```

9. Test at http://localhost/secure/. You should be prompted for a username and password.

## Authorisation by group

The above can be easily extended to do group authentication:

1. Create a groups file in /opt/apache/data/groups with this content:

```
administrators: elliot
```

2. Modify /var/www/htdocs/secure/.htaccess to authorise by group:

```
AuthType Basic
AuthName "Restricted Files"
AuthUserFile /opt/apache/data/passwords
AuthGroupFile /opt/apache/data/groups
Require group administrators
```

## Rewriting URLs

To demonstrate the use of other directives in .htaccess files, let's add a rewrite rule which redirects any request for files in the `secure` directory to the index page.

1. Load the rewrite module in httpd.conf:

```
LoadModule rewrite_module modules/mod_rewrite.so
```

2. Configure the secure directory for rewriting by modifying /opt/apache/conf/httpd.conf:

```
<Directory /var/www/htdocs/secure>
 Options SymLinksIfOwnerMatch
 AllowOverride FileInfo AuthConfig Limit
</Directory>
```

3. Add a rewrite directive to /var/www/htdocs/secure/.htaccess:

```
RewriteEngine On
RewriteRule .* index.php [L]
```

   This maps any URL of the form http://localhost/secure/xxxxx to http://localhost/secure/index.php.

4. Test in your browser.

There is a full guide to using mod_rewrite at:

http://httpd.apache.org/docs/2.2/misc/rewriteguide.html

# Virtual hosts

[Only about 1000 virtual hosts are possible per Apache instance using the approach detailed in this section. Beyond this limit, it is better to use an optimised solution like mod_vhost_alias instead.]

Virtual hosting allows "Running multiple websites on a single machine".

Two methods: IP-based or name-based

1. Name-based is simplest and requires fewer IP addresses (which are a scarce resource).
2. IP-based is more complex and needs one IP address for each host. For SSL sites on different hosts, must use IP-based hosting (can't have multiple SSL sites on a single IP address).

We're going to use name-based virtual hosts.

Our aim is to keep files related to an individual virtual host in one location reserved for that host; any core Apache log files etc. remain in a central location. This is the layout for each host:

- /var/www/jelica.com: base path for the virtual host
- /var/www/jelica.com/data (private web server/application data - e.g. things like passwords for PHP applications, web server password files generated using the htpasswd command, or SQLite database files)
- /var/www/jelica.com/htdocs (public files, PHP scripts, HTML)
- /var/www/jelica.com/cgi-bin (publicly-accessible CGI scripts)
- /var/www/jelica.com/log (logs for this host)
- /var/www/jelica.com/tmp (temporary files, e.g. files uploaded using PHP)

In cases where we are using chrooting, we might also have the following:

- /var/www/jelica.com/bin (private binaries executed by this host; allows us to isolate different binaries for different hosts, e.g. if one host requires PHP 5 and another wants PHP 4)

We'll miss this last one out of our virtual host configuration, for simplicity's sake.

We are also going to store each virtual host configuration in its own configuration file, named after the host. For example, for our jelica.com and oceanarea.com hosts, we will put their configuration in these two files:

1. /opt/apache/conf/jelica.com.conf
2. /opt/apache/conf/oceanarea.com.conf

I am not going to cover how to setup a machine to restrict a user to their own virtual host directories, with no access to the rest of the filesystem. (See the earlier section on chroot.)

## *Setting up jelica.com*

1. Create the user in charge of the domain:

   useradd --home /var/www/jelica.com jelicacom

2. Make the user's home directory accessible to Apache:

   chgrp apache /var/www/jelica.com
   chmod g+x /var/www/jelica.com

3. Create an htdocs directory for the user inside their home directory:

   mkdir /var/www/jelica.com/htdocs
   chown jelicacom:apache /var/www/jelica.com/htdocs
   chmod 2750 /var/www/jelica.com/htdocs

   Note that the last command also changes the sticky bit on the directory (the '2' at the start of the argument to chmod), so that any files added to the directory end up being owned by the apache group.
4. Make an index file for the domain in /var/www/jelica.com/index.php
5. Create the configuration file for the domain in /opt/apache/conf/jelica.com.conf

```
<VirtualHost *:80>
  DocumentRoot /var/www/jelica.com/htdocs
  ServerName jelica.com

  <Directory /var/www/jelica.com/htdocs>
  Order Allow,Deny
  Allow from all
  </Directory>
</VirtualHost>
```

6. Set permissions:

   chmod 600 /opt/apache/conf/jelica.com.conf
7. Add the directive to make Apache attach virtual host definitions to all IP addresses of the server:

```
NameVirtualHost *:80
```

   If you had a machine with multiple IP addresses, you could just set up one or two of these to serve virtual hosts from, e.g.

   NameVirtualHost 11.12.13.14:80
8. Pull the jelica.com configuration file into httpd.conf:

```
Include /opt/apache/conf/jelica.com.conf
```

9. Create a file in /home/jelicacom/htdocs for testing called index.php
10. Add an entry to /etc/hosts to map the domain name jelica.com to the localhost IP address. This enables to test our new virtual host without having to register the domain name etc..

    127.0.0.1 jelica.com
11. Test at http://jelica.com/
12. Test user login by attempting to login via ssh:

    ssh jelicacom@localhost

    Make sure the logged in user ends up in the /var/www/jelica.com directory.

## Setting up logging and CGI for a virtual host

We can setup the logs and CGI for the virtual host like this:

1. Make directories for the logs and CGI scripts inside the virtual host's directory:

   mkdir /var/www/jelica.com/logs
   mkdir /var/www/jelica.com/cgi-bin

2. Set permissions on the directories:

   chown -R jelicacom:apache /var/www/jelica.com
   chmod 2770 /var/www/jelica.com/logs
   chmod 2750 /var/www/jelica.com/cgi-bin

   Note the cgi-bin is set up the same as the htdocs directory. However, the logs directory is setup to allow the apache user to write into the directory.

3. Add these directives to jelica.com.conf, inside the <VirtualHost> directive:

```
<VirtualHost *:80>
  DocumentRoot /var/www/jelica.com/htdocs
  ServerName jelica.com


  <Directory /var/www/jelica.com/htdocs>
  Order Allow,Deny
  Allow from all
  </Directory>

  # error log
  ErrorLog /var/www/jelica.com/logs/error_log


  # access log
  <IfModule log_config_module>
    CustomLog /var/www/jelica.com/logs/access_log combined
  </IfModule>


  # cgi-bin
  <Directory /var/www/jelica.com/cgi-bin>
    Order Allow,Deny
    Allow from all
  </Directory>
  ScriptAlias /cgi-bin/ /var/www/jelica.com/cgi-bin/
</VirtualHost>
```

Note that log rotation will need to take the new log location into account; alternatively, you can leave it up to users to do their own log rotation.

## Allow following of symlinks

It is sometimes useful for users to be able to setup directories outside their main webroot but be able to symlink those directories into the webroot (this is useful for setting up Rails applications, for example).

You can turn this option on by adding this directive inside the <Directory /var/www/jelica.com/htdocs> directive:

```
Options SymLinksIfOwnerMatch
```

## Allowing directive overrides

We can also allow users to create authorisation and authentication for their directory inside .htaccess files by adding this directive inside the <Directory /var/www/jelica.com/htdocs> directive:

```
AllowOverride AuthConfig Limit FileInfo
```

We haven't overridden Indexes or Options for the directory. Allowing directives from these groups to be overridden can introduce security problems. However, you may have to if your users demand it.

The FileInfo setting allows users to employ mod_rewrite directives inside their directives (useful for friendly URL generation and commonly used by content management systems).

## Virtual host PHP configuration

We can use Apache configuration directives to affect how PHP behaves on a per-virtual-host basis. This can be used to secure what PHP running under that virtual host is able to do, and to put any PHP-related information (e.g. logs and session files) inside the virtual host directory.

There are two specific directives available to Apache for configuring PHP:

1. php_admin_flag <php_directive> <on_or_off>
   This is used to set boolean (true/false) PHP directives
2. php_admin_value <php_directive> <setting>
   This is used to set PHP directives which are non-boolean, e.g. strings, numbers

We can use these directives as follows:

1. `php_admin_value open_basedir /var/www/jelica.com`
   This restricts PHP to opening files inside the specified directory. Any attempt to open a file outside this directory will throw an error. Note that we've set this to the root for the virtual host, rather than the htdocs directory, to allow PHP access to the /data directory and the /tmp directory.
2. `php_admin_value error_log "/var/www/jelica.com/logs/php_log"`
   Put the PHP error log inside the virtual host's log directory
3. `php_admin_value session.save_path "/var/www/jelica.com/sessions"`
   `php_admin_value session.referer_check jelica.com`
   Put any session information into a separate sessions directory specific to this virtual host. This prevents users from other virtual hosts spying on this host's session data.

We also have to set the session.referer_check to jelica.com. However, if we are allowing domain parking, we might want to remove this constraint: if a cookie is set under the parked domain, the referer (when the cookie is passed to the next page) will reference the parked domain, causing PHP to reject the cookie (as the referer is wrong).

For the above to work, we will need a sessions directory:

mkdir /var/www/jelica.com/sessions
chown jelicacom.apache /var/www/jelica.com/sessions
chmod 2770 /var/www/jelica.com/sessions

4. `php_admin_flag file_uploads on`
   `php_admin_flag upload_tmp_dir /var/www/jelica.com/tmp`
   These settings allow users to upload files using their PHP scripts.

   Again, you will need a tmp directory for the virtual host:

   mkdir /var/www/jelica.com/tmp
   chown jelicacom.apache /var/www/jelica.com/tmp
   chmod 2770 /var/www/jelica.com/tmp

5. One more useful trick is to enable users to create files from inside their PHP scripts. For now, we will enable PHP to write only into the htdocs directory.

   The first step is to allow the apache user to write to the htdocs directory:

   chmod g+w /var/www/jelica.com/htdocs

   The only issue with allowing apache to create files inside htdocs is that the files created this way will not be editable by the virtual host's owner (in this case, jelicacom). One solution is to add the user to the apache group:

   usermod -G jelicacom,apache jelicacom

   However, this could potentially give the user access to files in other virtual hosts (i.e. any file owned by the apache group).

## *The final configuration file for our virtual host*

Combining these settings together inside the <Directory> setting for the virtual host (in /opt/apache/conf/jelica.com.conf) gives us:

```
<VirtualHost *:80>
  DocumentRoot /var/www/jelica.com/htdocs
  ServerName jelica.com

  <IfModule php5_module>
    php_admin_value open_basedir /var/www/jelica.com
```

```
    php_admin_value error_log /var/www/jelica.com/logs/php_log
    php_admin_value session.save_path /var/www/jelica.com/sessions
    php_admin_value session.referer_check jelica.com
    php_admin_flag file_uploads on
    php_admin_value upload_tmp_dir /var/www/jelica.com/tmp
  </IfModule>

  <Directory /var/www/jelica.com/htdocs>
    Order Allow,Deny
    Allow from all
    Options SymLinksIfOwnerMatch
    AllowOverride AuthConfig Limit FileInfo
  </Directory>

  # error log
  LogLevel info
  ErrorLog /var/www/jelica.com/logs/error_log

  # access log
  <IfModule log_config_module>
    CustomLog /var/www/jelica.com/logs/access_log combined
  </IfModule>

  # cgi-bin
  <Directory /var/www/jelica.com/cgi-bin>
    Order Allow,Deny
    Allow from all
  </Directory>

  ScriptAlias /cgi-bin/ /var/www/jelica.com/cgi-bin/
</VirtualHost>
```

## *Fixing localhost*

While this approach sets up jelica.com, it simultaneously destroys the configuration for localhost. To keep our current localhost settings, we need to remove the settings specific to the localhost website into a separate config. file called /opt/apache/conf/localhost.conf (remember to chmod to 600).

```
<VirtualHost *:80>
  ServerName localhost
```

```
  ### location of the web server document store
  DocumentRoot /var/www/htdocs

  ### logging

  # access log
  <IfModule log_config_module>
    CustomLog /var/log/apache/access_log combined
  </IfModule>

  ### base website
  <Directory /var/www/htdocs>
    Order Allow,Deny
    Allow from all
  </Directory>

  ### CGI
  <Directory /var/www/cgi-bin>
    Order Allow,Deny
    Allow from all
  </Directory>

  ScriptAlias /cgi-bin/ /var/www/cgi-bin/
</VirtualHost>
```

Note that I've left configuration for the error_log in the main httpd.conf file, as otherwise we get errors for localhost in /var/log/apache/error_log, and other generic error messages not related to localhost but to the server more generally in /opt/apache/log/error_log.

We can pull this configuration into httpd.conf using:

```
Include /opt/apache/conf/localhost.conf
```

Another pain is that this will also break our SSL config., as we have moved the <Directory> directive from httpd.conf and put it into localhost.conf. We can fix this by specifying a <Directory> directive inside the <VirtualHost> directive in /opt/apache/conf/ssl.conf like this:

```
### base website
<Directory /var/www/htdocs>
```

```
  Order Allow,Deny
  Allow from all
</Directory>
```

# Troubleshooting

Occasionally, we may run into problems with our Apache and PHP configuration. Here are a few ways to track down those problems.

## *Logs*

The log files should be your first port of call.

- **/var/log/apache/error_log**
  This will record any Apache-specific error messages, and also some other info. (e.g. SSL initialisation info.).
- **/var/log/apache/access_log**
  This records requests made to the web server. It is not as useful as the error_log, but can enable you to determine what was happening when an error occurred.
- **/var/log/php/php_log**
  Contains messages specific to PHP applications.

If the Apache log files are not sufficiently verbose, you can turn up the level of error reporting in httpd.conf by setting the LogLevel directive to "debug" (you need to restart for this to take effect).

## *Status reports*

Apache has a status module which enables you to get some general information about the server. However, it can be a security liability, as once the module is active, any user can add a status report to their virtual host via a .htaccess file. So we have disabled it.

Here's a sample configuration for reference:

```
### server status
LoadModule status_module modules/mod_status.so
<IfModule status_module>
  ExtendedStatus On

  <Location /status>
  SetHandler server-status

  Order Deny,Allow
  Deny from all
  Allow from 127.0.0.1
  </Location>
</IfModule>
```

This makes the status report available at http://localhost/status, but only to users on the local machine.

For more information, see:

http://httpd.apache.org/docs/2.2/mod/mod_status.html

## Standard tools

Some standard command line tools can give you a view of the system processes, which might help you find memory hogs and other system-level problems:

- **top**
  View processes and their memory and CPU usage in real time.
- **ps**
  View a snapshot of processes currently running.

## More advanced tools

Other debugging tools go outside the realms of Apache-specific and into more general system tools:

- **ethereal**
  A general network monitoring tool, which can be useful for viewing the responses returned by Apache.
- **strace**
  This gives a low-level view of the activity taking place when a binary executes, but its output is difficult to decipher. I've never found a need to use a tool with this level of complexity when debugging.

  As an example, you can monitor Apache startup using:

  ```
  strace /opt/apache/bin/httpd -k start
  ```

  This shows the files Apache attempts to read/write, and may help identify issues like missing system libraries.

# License



If you have any corrections/comments on this text, please feel free to contact me (elliot at moochlabs.com).