The Complete Guide to Securing Your Apache Web Server

Apache Security

O'REILLY®

Ivan Ristic

CHAPTER 2

Installation and Configuration

Installation is the first step in making Apache functional. Before you begin, you should have a clear idea of the installation's purpose. This idea, together with your paranoia level, will determine the steps you will take to complete the process. The system-hardening matrix (described in Chapter 1) presents one formal way of determining the steps. Though every additional step you make now makes the installation more secure, it also increases the time you will spend maintaining security. Think about it realistically for a moment. If you cannot put in that extra time later, then why bother putting the extra time in now? Don't worry about it too much, however. These things tend to sort themselves out over time: you will probably be eager to make everything perfect in the first couple of Apache installations you do; then, you will likely back off and find a balance among your security needs, the effort required to meet those needs, and available resources.

As a rule of thumb, if you are building a high profile web server—public or not—always go for a highly secure installation.

Though the purpose of this chapter is to be a comprehensive guide to Apache installation and configuration, you are encouraged to read others' approaches to Apache hardening as well. Every approach has its unique points, reflecting the personality of its authors. Besides, the opinions presented here are heavily influenced by the work of others. The Apache reference documentation is a resource you will go back to often. In addition to it, ensure you read the Apache Benchmark, which is a well-documented reference installation procedure that allows security to be quantified. It includes a semi-automated scoring tool to be used for assessment.

The following is a list of some of the most useful Apache installation documentation I have encountered:

- Apache Online Documentation (*http://httpd.apache.org/docs-2.0/*)
- Apache Security Tips (*http://httpd.apache.org/docs-2.0/misc/security_tips.html*)
- Apache Benchmark (*http://www.cisecurity.org/bench_apache.html*)

- “Securing Apache: Step-by-Step” by Artur Maj (*http://www.securityfocus.com/printable/infocus/1694*)
- “Securing Apache 2: Step-by-Step” by Artur Maj (*http://www.securityfocus.com/printable/infocus/1786*)

Installation

The installation instructions given in this chapter are designed to apply to both active branches (1.x and 2.x) of the Apache web server running on Linux systems. If you are running some other flavor of Unix, I trust you will understand what the minimal differences between Linux and your system are. The configuration advice given in this chapter works well for non-Unix platforms (e.g., Windows) but the differences in the installation steps are more noticeable:

- Windows does not offer the chroot functionality (see the section “Putting Apache in Jail”) or an equivalent.
- You are unlikely to install Apache on Windows from source code. Instead, download the binaries from the main Apache web site.
- Disk paths are different though the meaning is the same.

Source or Binary

One of the first decisions you will make is whether to compile the server from the source or use a binary package. This is a good example of the dilemma I mentioned at the beginning of this chapter. There is no one correct decision for everyone or one correct decision for you alone. Consider some pros and cons of the different approaches:

- By compiling from source, you are in the position to control everything. You can choose the compile-time options and the modules, and you can make changes to the source code. *This process will consume a lot of your time*, especially if you measure the time over the lifetime of the installation (it is the only correct way to measure time) and if you intend to use modules with frequent releases (e.g., PHP).
- Installation and upgrade is a breeze when binary distributions are used now that many vendors have tools to have operating systems updated automatically. You exchange some control over the installation in return for not having to do everything yourself. However, this choice means you will have to wait for security patches or for the latest version of your favorite module. In fact, the latest version of Apache or your favorite module may never come since most vendors choose to use one version in a distribution and only issue patches to that version to fix potential problems. This is a standard practice, which vendors use to produce stable distributions.

- The Apache version you intend to use will affect your decision. For example, nothing much happens in the 1.x branch, but frequent releases (with significant improvements) occur in the 2.x branch. Some operating system vendors have moved on to the 2.x branch, yet others remain faithful to the proven and trusted 1.x branch.

> The Apache web server is a victim of its own success. The web server from the 1.x branch works so well that many of its users have no need to upgrade. In the long term this situation only slows down progress because developers spend their time maintaining the 1.x branch instead of adding new features to the 2.x branch. Whenever you can, use Apache 2!

This book shows the approach of compiling from the source code since that approach gives us the most power and the flexibility to change things according to our taste. To download the source code, go to *http://httpd.apache.org* and pick the latest release of the branch you want to use.

Downloading the source code

Habitually checking the integrity of archives you download from the Internet is a good idea. The Apache distribution system works through mirrors. Someone may decide to compromise a mirror and replace the genuine archive with a trojaned version (a version that feels like the original but is modified in some way, for example, programmed to allow the attacker unlimited access to the web server). You will go through a lot of trouble to secure your Apache installation, and it would be a shame to start with a compromised version.

If you take a closer look at the Apache download page, you will discover that though archive links point to mirrors, archive signature links always point to the main Apache web site.

One way to check the integrity is to calculate the MD5 sum of the archive and to compare it with the sum in the signature file. An MD5 sum is an example of a hash function, also known as one-way encryption (see Chapter 4 for further information). The basic idea is that, given data (such as a binary file), a hash function produces seemingly random output. However, the output is always the same when the input is the same, and it is not possible to reconstruct the input given the output. In the example below, the first command calculates the MD5 sum of the archive that was downloaded, and the second command downloads and displays the contents of the MD5 sum from the main Apache web site. You can see the sums are identical, which means the archive is genuine:

```
$ md5sum httpd-2.0.50.tar.gz
8b251767212aebf41a13128bb70c0b41  httpd-2.0.50.tar.gz
$ wget -O - -q http://www.apache.org/dist/httpd/httpd-2.0.50.tar.gz.md5
8b251767212aebf41a13128bb70c0b41  httpd-2.0.50.tar.gz
```

Using MD5 sums to verify archive integrity can be circumvented if an intruder compromises the main distribution site. He will be able to replace the archives and the signature files, making the changes undetectable.

A more robust, but also a more complex approach is to use *public-key cryptography* (described in detail in Chapter 4) for integrity validation. In this approach, Apache developers use their cryptographic keys to sign the distribution digitally. This can be done with the help of GnuPG, which is installed on most Unix systems by default. First, download the PGP signature for the appropriate archive, such as in this example:

```
$ wget http://www.apache.org/dist/httpd/httpd-2.0.50.tar.gz.asc
```

Attempting to verify the signature at this point will result in GnuPG complaining about not having the appropriate key to verify the signature:

```
$ gpg httpd-2.0.50.tar.gz.asc
gpg: Signature made Tue 29 Jun 2004 01:14:14 AM BST using DSA key ID DE885DD3
gpg: Can't check signature: public key not found
```

GnuPG gives out the unique key ID (`DE885DD3`), which can be used to fetch the key from one of the key servers (for example, `pgpkeys.mit.edu`):

```
$ gpg --keyserver pgpkeys.mit.edu --recv-key DE885DD3
gpg: /home/ivanr/.gnupg/trustdb.gpg: trustdb created
gpg: key DE885DD3: public key "Sander Striker <striker@apache.org>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

This time, an attempt to check the signature gives satisfactory results:

```
$ gpg httpd-2.0.50.tar.gz.asc
gpg: Signature made Tue 29 Jun 2004 01:14:14 AM BST using DSA key ID DE885DD3
gpg: Good signature from "Sander Striker <striker@apache.org>"
gpg:                 aka "Sander Striker <striker@striker.nl>"
gpg:                 aka "Sander Striker <striker@striker.nl>"
gpg:                 aka "Sander Striker <striker@apache.org>"
gpg: checking the trustdb
gpg: no ultimately trusted keys found
Primary key fingerprint: 4C1E ADAD B4EF 5007 579C  919C 6635 B6C0 DE88 5DD3
```

At this point, we can be confident the archive is genuine. On the Apache web site, a file contains the public keys of all Apache developers (*http://www.apache.org/dist/httpd/KEYS*). You can use it to import all their keys at once but I prefer to download keys from a third-party key server. You should ignore the suspicious looking message ("no ultimately trusted keys found") for the time being. It is related to the concept of *web of trust* (covered in Chapter 4).

Downloading patches

Sometimes, the best version of Apache is not contained in the most recent version archive. When a serious bug or a security problem is discovered, Apache developers will fix it quickly. But getting a new revision of the software release takes time because of the additional full testing overhead required. Sometimes, a problem is not considered serious enough to warrant an early next release. In such cases, source code patches are made available for download at *http://www.apache.org/dist/httpd/patches/*. Therefore, the complete source code download procedure consists of downloading the latest official release followed by a check for and possible download of optional patches.

Static Binary or Dynamic Modules

The next big decision is whether to create a single static binary, or to compile Apache to use dynamically loadable modules. Again, the tradeoff is whether to spend more time in order to get more security.

- Static binary is reportedly faster. If you want to squeeze the last bit of performance out of your server, choose this option. But, as hardware is becoming faster and faster, the differences between the two versions will no longer make a difference.
- A static server binary cannot have a precompiled dynamic module *backdoor* added to it. (If you are unfamiliar with the concept of backdoors, see the sidebar "Apache Backdoors.") Adding a backdoor to a dynamically compiled server is as simple as including a module into the configuration file. To add a backdoor to a statically compiled server, the attacker has to recompile the whole server from scratch.
- With a statically linked binary, you will have to reconfigure and recompile the server every time you want to change a single module.
- The static version may use more memory depending on the operating system used. One of the points of having a dynamic library is to allow the operating system to load the library once and reuse it among active processes. Code that is part of a statically compiled binary cannot be shared in this way. Some operating systems, however, have a memory usage reduction feature, which is triggered when a new process is created by duplication of an existing process (known as *forking*). This feature, called *copy-on-write*, allows the operating system to share the memory in spite of being statically compiled. The only time the memory will be duplicated is when one of the processes attempts to change it. Linux and FreeBSD support copy-on-write, while Solaris reportedly does not.

Apache Backdoors

For many systems, a web server on port 80 is the only point of public access. So, it is no wonder black hats have come up with ideas of how to use this port as their point of entry into the system. A *backdoor* is malicious code that can give direct access to the heart of the system, bypassing normal access restrictions. An example of a backdoor is a program that listens on a high port of a server, giving access to anyone who knows the special password (and not to normal system users). Such backdoors are easy to detect provided the server is routinely scanned for open ports: a new open port will trigger all alarm bells.

Apache backdoors do not need to open new ports since they can reuse the open port 80. A small fragment of code will examine incoming HTTP requests, opening "the door" to the attacker when a specially crafted request is detected. This makes Apache backdoors stealthy and dangerous.

A quick search on the Internet for "apache backdoor" yields three results:

- *http://packetstormsecurity.org/UNIX/penetration/rootkits/apachebd.tgz*
- *http://packetstormsecurity.org/advisories/b0f/mod_backdoor.c*
- *http://packetstormsecurity.org/web/mod_rootme-0.2.tgz*

The approach in the first backdoor listed is to patch the web server itself, which requires the Apache source code and a compiler to be available on the server to allow for recompilation. A successful exploitation gives the attacker a root shell on the server (assuming the web server is started as *root*), with no trace of the access in the log files.

The second link is for a dynamically loadable module that appends itself to an existing server. It allows the attacker to execute a shell command (as the web server user) sent to the web server as a single, specially crafted `GET` request. This access will be logged but with a faked entry for the home page of the site, making it difficult to detect.

The third link is also for a dynamically loadable module. To gain *root* privileges this module creates a special process when Apache starts (Apache is still running as *root* at that point) and uses this process to perform actions later.

The only reliable way to detect a backdoor is to use host intrusion detection techniques, discussed in Chapter 9.

Folder Locations

In this chapter, I will assume the following locations for the specified types of files:

Binaries and supporting files
: */usr/local/apache*

Public files
: */var/www/htdocs* (this directory is referred to throughout this book as the web server tree)

Private web server or application data
: */var/www/data*

Publicly accessible CGI scripts
: */var/www/cgi-bin*

Private binaries executed by the web server
: */var/www/bin*

Log files
: */var/www/logs*

Installation locations are a matter of taste. You can adopt any layout you like as long as you use it consistently. Special care must be taken when deciding where to store the log files since they can grow over time. Make sure they reside on a partition with enough space and where they won't jeopardize the system by filling up the root partition.

Different circumstances dictate different directory layouts. The layout used here is suitable when only one web site is running on the web server. In most cases, you will have many sites per server, in which case you should create a separate set of directories for each. For example, you might create the following directories for one of those sites:

/var/www/apachesecurity.net/bin
/var/www/apachesecurity.net/cgi-bin
/var/www/apachesecurity.net/data
/var/www/apachesecurity.net/htdocs
/var/www/apachesecurity.net/logs

A similar directory structure would exist for another one of the sites:

/var/www/modsecurity.org/bin
/var/www/modsecurity.org/cgi-bin
/var/www/modsecurity.org/data
/var/www/modsecurity.org/htdocs
/var/www/modsecurity.org/logs

Installation Instructions

Before the installation can take place Apache must be made aware of its environment. This is done through the *configure* script:

```
$ ./configure --prefix=/usr/local/apache
```

The *configure* script explores your operating system and creates the *Makefile* for it, so you can execute the following to start the actual compilation process, copy the files into the directory set by the `--prefix` option, and execute the *apachectl* script to start the Apache server:

```
$ make
# make install
# /usr/local/apache/bin/apachectl start
```

Though this will install and start Apache, you also need to configure your operating system to start Apache when it boots. The procedure differs from system to system on Unix platforms but is usually done by creating a symbolic link to the *apachectl* script for the relevant *runlevel* (servers typically use run level 3):

```
# cd /etc/rc3.d
# ln -s /usr/local/apache/bin/apachectl S85httpd
```

On Windows, Apache is configured to start automatically when you install from a binary distribution, but you can do it from a command line by calling Apache with the `-k install` command switch.

Testing the installation

To verify the startup has succeeded, try to access the web server using a browser as a client. If it works you will see the famous "Seeing this instead of the website you expected?" page, as shown in Figure 2-1. At the time of this writing, there are talks on the Apache developers' list to reduce the welcome message to avoid confusing users (not administrators but those who stumble on active but unused Apache installations that are publicly available on the Internet).

Figure 2-1. Apache post-installation welcome page

As a bonus, toward the end of the page, you will find a link to the Apache reference manual. If you are near a computer while reading this book, you can use this copy of the manual to learn configuration directive specifics.

Using the *ps* tool, you can find out how many Apache processes there are:

```
$ ps -Ao user,pid,ppid,cmd | grep httpd
root     31738     1  /usr/local/apache/bin/httpd -k start
httpd    31765 31738  /usr/local/apache/bin/httpd -k start
httpd    31766 31738  /usr/local/apache/bin/httpd -k start
httpd    31767 31738  /usr/local/apache/bin/httpd -k start
httpd    31768 31738  /usr/local/apache/bin/httpd -k start
httpd    31769 31738  /usr/local/apache/bin/httpd -k start
```

Using *tail*, you can see what gets logged when different requests are processed. Enter a nonexistent filename in the browser location bar and send the request to the web server; then examine the access log (logs are in the */var/www/logs* folder). The example below shows successful retrieval (as indicated by the 200 return status code) of a file that exists, followed by an unsuccessful attempt (404 return status code) to retrieve a file that does not exist:

```
192.168.2.3 - - [21/Jul/2004:17:12:22 +0100] "GET /manual/images/feather.gif
HTTP/1.1" 200 6471
192.168.2.3 - - [21/Jul/2004:17:20:05 +0100] "GET /manual/not-here
HTTP/1.1" 404 311
```

Here is what the error log contains for this example:

```
[Wed Jul 21 17:17:04 2004] [notice] Apache/2.0.50 (Unix) configured
-- resuming normal operations
[Wed Jul 21 17:20:05 2004] [error] [client 192.168.2.3] File does not
exist: /usr/local/apache/manual/not-here
```

The idea is to become familiar with how Apache works. As you learn what constitutes normal behavior, you will learn how to spot unusual events.

Selecting modules to install

The theory behind module selection says that the smaller the number of modules running, the smaller the chances of a vulnerability being present in the server. Still, I do not think you will achieve much by being too strict with default Apache modules. The likelihood of a vulnerability being present in the code rises with the complexity of the module. Chances are that the really complex modules, such as *mod_ssl* (and the OpenSSL libraries behind it), are the dangerous ones.

Your strategy should be to identify the modules you need to have as part of an installation and not to include anything extra. Spend some time researching the modules distributed with Apache so you can correctly identify which modules are needed and which can be safely turned off. The complete module reference is available at *http://httpd.apache.org/docs-2.0/mod/*.

The following modules are more dangerous than the others, so you should consider whether your installation needs them:

mod_userdir
: Allows each user to have her own web site area under the *~username* alias. This module could be used to discover valid account usernames on the server because Apache responds differently when the attempted username does not exist (returning status `404`) and when it does not have a special web area defined (returning `403`).

mod_info
: Exposes web server configuration as a web page.

mod_status
: Provides real-time information about Apache, also as a web page.

mod_include
: Provides simple scripting capabilities known under the name *server-side includes* (SSI). It is very powerful but often not used.

On the other hand, you should include these modules in your installation:

mod_rewrite
: Allows incoming requests to be rewritten into something else. Known as the "Swiss Army Knife" of modules, you will need the functionality of this module.

mod_headers
: Allows request and response headers to be manipulated.

mod_setenvif
: Allows environment variables to be set conditionally based on the request information. Many other modules' conditional configuration options are based on environment variable tests.

In the *configure* example, I assumed acceptance of the default module list. In real situations, this should rarely happen as you will want to customize the module list to your needs. To obtain the list of modules activated by default in Apache 1, you can ask the *configure* script. I provide only a fragment of the output below, as the complete output is too long to reproduce in a book:

```
$ ./configure --help
...
[access=yes      actions=yes     alias=yes      ]
[asis=yes        auth_anon=no    auth_dbm=no    ]
[auth_db=no      auth_digest=no  auth=yes       ]
[autoindex=yes   cern_meta=no    cgi=yes        ]
[digest=no       dir=yes         env=yes        ]
[example=no      expires=no      headers=no     ]
[imap=yes        include=yes     info=no        ]
[log_agent=no    log_config=yes  log_forensic=no]
[log_referer=no  mime_magic=no   mime=yes       ]
[mmap_static=no  negotiation=yes proxy=no       ]
```

```
[rewrite=no      setenvif=yes    so=no          ]
[speling=no      status=yes      unique_id=no   ]
[userdir=yes     usertrack=no    vhost_alias=no ]
...
```

As an example of interpreting the output, `userdir=yes` means that the module *mod_userdir* will be activated by default. Use the `--enable-module` and `--disable-module` directives to adjust the list of modules to be activated:

```
$ ./configure \
> --prefix=/usr/local/apache \
> --enable-module=rewrite \
> --enable-module=so \
> --disable-module=imap \
> --disable-module=userdir
```

Obtaining a list of modules activated by default in Apache 2 is more difficult. I obtained the following list by compiling Apache 2.0.49 without passing any parameters to the *configure* script and then asking the *httpd* binary to produce a list of modules:

```
$ ./httpd -l
Compiled in modules:
  core.c
  mod_access.c
  mod_auth.c
  mod_include.c
  mod_log_config.c
  mod_env.c
  mod_setenvif.c
  prefork.c
  http_core.c
  mod_mime.c
  mod_status.c
  mod_autoindex.c
  mod_asis.c
  mod_cgi.c
  mod_negotiation.c
  mod_dir.c
  mod_imap.c
  mod_actions.c
  mod_userdir.c
  mod_alias.c
  mod_so.c
```

To change the default module list on Apache 2 requires a different syntax than that used on Apache 1:

```
$ ./configure \
> --prefix=/usr/local/apache \
> --enable-rewrite \
> --enable-so \
> --disable-imap \
> --disable-userdir
```

Configuration and Hardening

Now that you know your installation works, make it more secure. Being brave, we start with an empty configuration file, and work our way up to a fully functional configuration. Starting with an empty configuration file is a good practice since it increases your understanding of how Apache works. Furthermore, the default configuration file is large, containing the directives for everything, including the modules you will never use. It is best to keep the configuration files nice, short, and tidy.

Start the configuration file (*/usr/local/apache/conf/httpd.conf*) with a few general-purpose directives:

```
# location of the web server files
ServerRoot /usr/local/apache
# location of the web server tree
DocumentRoot /var/www/htdocs
# path to the process ID (PID) file, which
# stores the PID of the main Apache process
PidFile /var/www/logs/httpd.pid
# which port to listen at
Listen 80
# do not resolve client IP addresses to names
HostNameLookups Off
```

Setting Up the Server User Account

Upon installation, Apache runs as a user *nobody*. While this is convenient (this account normally exists on all Unix operating systems), it is a good idea to create a separate account for each different task. The idea behind this is that if attackers break into the server through the web server, they will get the privileges of the web server. The intruders will have the same priveleges as in the user account. By having a separate account for the web server, we ensure the attackers do not get anything else free.

The most commonly used username for this account is *httpd*, and some people use *apache*. We will use the former. Your operating system may come pre-configured with an account for this purpose. If you like the name, use it; otherwise, delete it from the system (e.g., using the *userdel* tool) to avoid confusion later. To create a new account, execute the following two commands while running as *root*.

```
# groupadd httpd
# useradd httpd -g httpd -d /dev/null -s /sbin/nologin
```

These commands create a group and a user account, assigning the account the home directory */dev/null* and the shell */sbin/nologin* (effectively disabling login for the account). Add the following two lines to the Apache configuration file *httpd.conf*:

```
User httpd
Group httpd
```

Setting Apache Binary File Permissions

After creating the new user account your first impulse might be to assign ownership over the Apache installation to it. I see that often, but do not do it. For Apache to run on port 80, it must be started by the user *root*. Allowing any other account to have write access to the *httpd* binary would give that account privileges to execute anything as *root*.

This problem would occur, for example, if an attacker broke into the system. Working as the Apache user (*httpd*), he would be able to replace the *httpd* binary with something else and shut the web server down. The administrator, thinking the web server had crashed, would log in and attempt to start it again and would have fallen into the trap of executing a Trojan program.

That is why we make sure only *root* has write access:

```
# chown -R root:root /usr/local/apache
# find /usr/local/apache -type d | xargs chmod 755
# find /usr/local/apache -type f | xargs chmod 644
```

No reason exists why anyone else other than the *root* user should be able to read the Apache configuration or the logs:

```
# chmod -R go-r /usr/local/apache/conf
# chmod -R go-r /usr/local/apache/logs
```

Configuring Secure Defaults

Unless told otherwise, Apache will serve any file it can access. This is probably not what most people want; a configuration error could accidentally expose vital system files to anyone caring to look. To change this, we would deny access to the complete filesystem and then allow access to the document root only by placing the following directives in the *httpd.conf* configuration file:

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

Options directive

This sort of protection will not help with incorrectly or maliciously placed symbolic links that point outside the */var/www/htdocs* web server root. System users could create symbolic links to resources they do not own. If someone creates such a link and the web server can read the resource, it will accept a request to serve the resource to the public. Symbolic link usage and other file access restrictions are controlled with the Options directive (inside a <Directory> directive). The Options directive can have one or more of the following values:

All
: All options listed below except MultiViews. This is the default setting.

None
: None of the options will be enabled.

ExecCGI
: Allows execution of CGI scripts.

FollowSymLinks
: Allows symbolic links to be followed.

Includes
: Allows server-side includes.

IncludesNOEXEC
: Allows SSIs but not the exec command, which is used to execute external scripts. (This setting does not affect CGI script execution.)

Indexes
: Allows the server to generate the list of files in a directory when a default index file is absent.

MultiViews
: Allows content negotiation.

SymLinksIfOwnerMatch
: Allows symbolic links to be followed if the owner of the link is the same as the owner of the file it points to.

The following configuration directive will disable symbolic link usage in Apache:

```
Options -FollowSymLinks
```

The minus sign before the option name instructs Apache to keep the existing configuration and disable the listed option. The plus character is used to add an option to an existing configuration.

The Apache syntax for adding and removing options can be confusing. If *all* option names in a given Options statement for a particular directory are preceded with a plus or minus character, then the new configuration will be merged with the existing configuration, with the new configuration overriding the old values. In all other cases, the old values will be ignored, and only the new values will be used.

If you need symbolic links consider using the Alias directive, which tells Apache to incorporate an external folder into the web server tree. It serves the same purpose but is more secure. For example, it is used in the default configuration to allow access to the Apache manual:

```
Alias /manual/ /usr/local/apache/manual/
```

If you want to keep symbolic links, it is advisable to turn ownership verification on by setting the `SymLinksIfOwnerMatch` option. After this change, Apache will follow symbolic links if the target and the destination belong to the same user:

```
Options -FollowSymLinks +SymLinksIfOwnerMatch
```

Other features you do not want to allow include the ability to have scripts and server-side includes executed anywhere in the web server tree. Scripts should always be placed in special folders, where they can be monitored and controlled.

```
Options -Includes -ExecCGI
```

If you do not intend to use content negotiation (to have Apache choose a file to serve based on the client's language preference), you can (and should) turn all of these features off in one go:

```
Options None
```

Modules sometimes use the settings determined with the `Options` directive to allow or deny access to their features. For example, to be able to use *mod_rewrite* in per-directory configuration files, the `FollowSymLinks` option must be turned on.

AllowOverride directive

In addition to serving any file it can access by default, Apache also by default allows parts of configuration data to be placed under the web server tree, in files normally named *.htaccess*. Configuration information in such files can override the information in the *httpd.conf* configuration file. Though this can be useful, it slows down the server (because Apache is forced to check whether the file exists in any of the subfolders it serves) and allows anyone who controls the web server tree to have limited control of the web server. This feature is controlled with the `AllowOverride` directive, which, like `Options`, appears within the `<Directory>` directive specifying the directory to which the options apply. The `AllowOverride` directive supports the following options:

`AuthConfig`
: Allows use (in *.htaccess* files) of the authorization directives (explained in Chapter 7)

`FileInfo`
: Allows use of the directives controlling document types

`Indexes`
: Allows use of the directives controlling directory indexing

`Limit`
: Allows use of the directives controlling host access

`Options`
: Allows use of the directives controlling specific directory functions (the `Options` and `XbitHack directives`)

`All`
: Allows all options listed

`None`
: Ignores *.htaccess* configuration files

For our default configuration, we choose the `None` option. So, our `<Directory>` directives are now:

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

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

Modules sometimes use `AllowOverride` settings to make other decisions as to whether something should be allowed. Therefore, a change to a setting can have unexpected consequences. As an example, including `Options` as one of the `AllowOverride` options will allow PHP configuration directives to be used in *.htaccess* files. In theory, every directive of every module should fit into one of the `AllowOverride` settings, but in practice it depends on whether their respective developers have considered it.

Enabling CGI Scripts

Only enable CGI scripts when you need them. When you do, a good practice is to have all scripts grouped in a single folder (typically named *cgi-bin*). That way you will know what is executed on the server. The alternative solution is to enable script execution across the web server tree, but then it is impossible to control script execution; a developer may install a script you may not know about. To allow execution of scripts in the */var/www/cgi-bin* directory, include the following `<Directory>` directive in the configuration file:

```
<Directory /var/www/cgi-bin>
    Options ExecCGI
    SetHandler cgi-script
</Directory>
```

An alternative is to use the `ScriptAlias` directive, which has a similar effect:

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

There is a subtle but important difference between these two approaches. In the first approach, you are setting the configuration for a directory directly. In the second, a *virtual* directory is created and configured, and the original directory is still left without a configuration. In the examples above, there is no difference because the names of the two directories are the same, and the virtual directory effectively hides the real one. But if the name of the virtual directory is different (e.g., *my-cgi-bin/*), the real directory will remain visible under its own name and you would end up with one web site directory where files are treated like scripts (*my-cgi-bin/*) and with one where files are treated as files (*cgi-bin/*). Someone could download the source code of all scripts from the latter. Using the `<Directory>` directive approach is recommended when the directory with scripts is under the web server tree. In other cases, you may use `ScriptAlias` safely.

Logging

Having a record of web server activity is of utmost importance. Logs tell you which content is popular and whether your server is underutilized, overutilized, misconfigured, or misused. This subject is so important that a complete chapter is dedicated to it. Here I will only bring your attention to two details: explaining how to configure logging and how not to lose valuable information. It is not important to understand all of the meaning of logging directives at this point. When you are ready, proceed to Chapter 8 for a full coverage.

Two types of logs exist. The *access log* is a record of all requests sent to a particular web server or web site. To create an access log, you need two steps. First, use the `LogFormat` directive to define a logging format. Then, use the `CustomLog` directive to create an access log in that format:

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

The *error log* contains a record of all system events (such as web server startup and shutdown) and a record of errors that occurred during request processing. For example, a request for a resource that does not exist generates an HTTP `404` response for the client, one entry in the access log, and one entry in the error log. Two directives are required to set up the error log, just as for the access log. The following `LogLevel` directive increases the logging detail from a default value of `notice` to `info`. The `ErrorLog` directive creates the actual log file:

```
LogLevel info
ErrorLog /var/www/logs/error_log
```

Setting Server Configuration Limits

Though you are not likely to fine-tune the server during installation, you must be aware of the existence of server limits and the way they are configured. Incorrectly configured limits make a web server an easy target for attacks (see Chapter 5). The following configuration directives all show default Apache configuration values and define how long the server will wait for a slow client:

```
# wait up to 300 seconds for slow clients
TimeOut 300
# allow connections to be reused between requests
KeepAlive On
# allow a maximum of 100 requests per connection
MaxKeepAliveRequests 100
# wait up to 15 seconds for the next
# request on an open connection
KeepAliveTimeout 15
```

The default value for the connection timeout (300 seconds) is too high. You can safely reduce it below 60 seconds and increase your tolerance against *denial of service* (DoS) *attacks* (see Chapter 5).

The following directives impose limits on various aspects of an HTTP request:

```
# impose no limits on the request body
LimitRequestBody 0
# allow up to 100 headers in a request
LimitRequestFields 100
# each header may be up to 8190 bytes long
LimitRequestFieldsize 8190
# the first line of the request can be
# up to 8190 bytes long
LimitRequestLine 8190
# limit the XML request body to 1 million bytes(Apache 2.x only)
LimitXMLRequestBody 1000000
```

`LimitXMLRequestBody` is an Apache 2 directive and is used by the *mod_dav* module to limit the size of its command requests (which are XML-based).

Seeing that the maximal size of the request body is unlimited by default (2 GB in practice), you may wish to specify a more sensible value for `LimitRequestBody`. You can go as low as 64 KB if you do not plan to support file uploads in the installation.

The following directives control how server instances are created and destroyed in Apache 1 and sometimes in Apache 2 (as described further in the following text):

```
# keep 5 servers ready to handle requests
MinSpareServers 5
# do not keep more than 10 servers idle
MaxSpareServers 10
# start with 5 servers
StartServers 5
# allow a max of 150 clients at any given time
MaxClients 150
# allow unlimited requests per server
MaxRequestsPerChild 0
```

You may want to lower the maximal number of clients (`MaxClients`) if your server does not have enough memory to handle 150 Apache instances at one time.

You should make a habit of putting a limit on the maximal number of requests served by one server instance, which is unlimited by default in Apache 1 (as indicated by the `0` `MaxRequestsPerChild` value) but set to `10000` in Apache 2. When a server instance reaches the limit, it will be shut down and replaced with a fresh copy. A high value such as `1000` (or even more) will not affect web server operation but will help if an Apache module has a memory leak. Interestingly, when the Keep-Alive feature (which allows many requests to be performed over a single network connection) is used, all requests performed over a single Keep-Alive connection will be counted as one for the purposes of `MaxRequestsPerChild` handling.

Apache 2 introduces the concept of *multiprocessing modules* (MPMs), which are special-purpose modules that determine how request processing is organized. Only one MPM can be active at any one time. MPMs were introduced to allow processing to be optimized for each operating system individually. The Apache 1 processing model (multiple processes, no threads, each process handling one request at one time) is called *prefork,* and it is the default processing model in Apache 2 running on Unix platforms. On Windows, Apache always runs as a single process with multiple execution threads, and the MPM for that is known as *winnt*. On Unix systems running Apache 2, it is possible to use the *worker* MPM, which is a hybrid, as it supports many processes each with many threads. For the *worker* MPM, the configuration is similar to the following (refer to the documentation for the complete description):

```
# the maximum number of processes
ServerLimit 16
# how many processes to start with
StartServers 2
# how many threads per process to create
ThreadsPerChild 25
# minimum spare threads across all processes
MinSpareThreads 25
# maximum spare threads across all processes
MaxSpareThreads 75
# maximum clients at any given time
MaxClients 150
```

Since the number of threads per process is fixed, the Apache worker MPM will change the number of active processes to obey the minimum and maximum spare threads configured. Unlike with the *prefork* MPM, the `MaxClients` directive now controls the maximum number of active threads at any given time.

Preventing Information Leaks

By default, Apache provides several bits of information to anyone interested. Any information obtained by attackers helps them build a better view of the system and makes it easier for them to break into the system.

For example, the installation process automatically puts the email address of the user compiling Apache (or, rather, the email address it thinks is the correct email address) into the configuration file. This reveals the account to the public, which is undesirable. The following directive replaces the Apache-generated email address with a generic address:

```
ServerAdmin webmaster@apachesecurity.net
```

By default, the email address defined with this directive appears on server-generated pages. Since this is probably not what you want, you can turn off this feature completely via the following directive:

```
ServerSignature Off
```

The HTTP protocol defines a response header field `Server`, whose purpose is to identify the software responding to the request. By default, Apache populates this header with its name, version number, and names and version numbers of all its modules willing to identify themselves. You can see what this looks like by sending a test request to the newly installed server:

```
$ telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Fri, 19 Mar 2004 22:05:35 GMT
Server: Apache/1.3.29 (Unix)
Content-Location: index.html.en
Vary: negotiate,accept-language,accept-charset
TCN: choice
Last-Modified: Fri, 04 May 2001 00:00:38 GMT
ETag: "4002c7-5b0-3af1f126;405a21d7"
Accept-Ranges: bytes
Content-Length: 1456
Connection: close
Content-Type: text/html
Content-Language: en
Expires: Fri, 19 Mar 2004 22:05:35 GMT
```

This header field reveals specific and valuable information to the attacker. You can't hide it completely (this is not entirely true, as you will find in the next section), but you can tell Apache to disclose only the name of the server ("Apache").

```
ServerTokens ProductOnly
```

We turned off the directory indexing feature earlier when we set the `Options` directive to have the value `None`. Having the feature off by default is a good approach. You can enable it later on a per-directory basis:

```
<Directory /var/www/htdocs/download>
    Options +Indexes
</Directory>
```

Automatic directory indexes are dangerous because programmers frequently create folders that have no default indexes. When that happens, Apache tries to be helpful and lists the contents of the folder, often showing the names of files that are publicly available (because of an error) but should not be seen by anyone, such as the following:

- Files (usually archives) stored on the web server but not properly protected (e.g., with a password) because users thought the files could not be seen and thus were secure
- Files that were uploaded "just for a second" but were never deleted
- Source code backup files automatically created by text editors and uploaded to the production server by mistake
- Backup files created as a result of direct modification of files on the production server

To fight the problem of unintentional file disclosure, you should turn off automatic indexing (as described in the "AllowOverride directive" section) and instruct Apache to reject all requests for files matching a series of regular expressions given below. Similar configuration code exists in the default *httpd.conf* file to deny access to *.htaccess* files (the per-directory configuration files I mentioned earlier). The following extends the regular expression to look for various file extensions that should normally not be present on the web server:

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

The `FilesMatch` directive only looks at the last part of the full filename (the basename), and thus, `FilesMatch` configuration specifications do not apply to directory names. To completely restrict access to a particular directory, for example to deny access to CVS administrative files (frequently found on web sites), use something like:

```
<DirectoryMatch /CVS/>
    Order Allow,Deny
    Deny from all
</DirectoryMatch>
```

Changing Web Server Identity

One of the principles of web server hardening is hiding as much information from the public as possible. By extending the same logic, hiding the identity of the web server makes perfect sense. This subject has caused much controversy. Discussions usually start because Apache does not provide facilities to control all of the content provided in the `Server` header field, and some poor soul tries to influence Apache developers to add it. Because no clear technical reasons support either opinion, discussions continue.

I have mentioned the risks of providing server information in the `Server` response header field defined in the HTTP standard, so a first step in our effort to avoid this will be to fake its contents. As you will see later, this is often not straightforward, but it can be done. Suppose we try to be funny and replace our standard response "Apache/1.3.30 (Unix)" with "Microsoft-IIS/5.0" (it makes no difference to us that Internet Information Server has a worse security record than Apache; our goal is to hide who we are). An attacker sees this but sees no trace of Active Server Pages (ASP) on the server, and that makes him suspicious. He decides to employ *operating system fingerprinting*. This technique uses the variations in the implementations of the TCP/IP protocol to figure out which operating system is behind an IP address. This functionality comes with the popular network scanner NMAP. Running NMAP against a Linux server will sometimes reveal that the server is not running Windows. Microsoft IIS running on a Linux server—not likely!

There are also differences in the implementations of the HTTP protocol supplied by different web servers. *HTTP fingerprinting* exploits these differences to determine the make of the web server. The differences exist for the following reasons:

- Standards do not define every aspect of protocols. Some parts of the standard are merely recommendations, and some parts are often intentionally left vague because no one at the time knew how to solve a particular problem so it was left to resolve itself.
- Standards sometimes do not define trivial things.
- Developers often do not follow standards closely, and even when they do, they make mistakes.

The most frequently used example of web server behavior that may allow exploitation is certainly the way Apache treats URL encoded forward slash characters. Try this:

1. Open a browser window, and type in the address **`http://www.apachesecurity.net//`** (two forward slashes at the end). You will get the home page of the site.
2. Replace the forward slash at the end with `%2f` (the same character but URL-encoded): **`http://www.apachesecurity.net/%2f`**. The web server will now respond with a `404` (Not Found) response code!

This happens only if the site runs Apache. In two steps you have determined the make of the web server without looking at the `Server` header field. Automating this check is easy.

This behavior was so widely and frequently discussed that it led Apache developers to introduce a directive (`AllowEncodedSlashes`) to the 2.x branch to toggle how Apache behaves. This will not help us much in our continuing quest to fully control the content provided in the `Server` header field. There is no point in continuing to fight for this. In theory, the only way to hide the identity of the server is to put a reverse proxy (see Chapter 9) in front and instruct it to alter the order of header fields in the response, alter their content, and generally do everything possible to hide the server behind it. Even if someone succeeds at this, this piece of software will be so unique that the attacker will identify the reverse proxy successfully, which is as dangerous as what we have been trying to hide all along.

Not everything is lost, however. You may not be able to transform your installation's identity, but you can pretend to be, say, a different version of the same web server. Or you can pretend to be a web server with a list of modules different from reality. There is a great opportunity here to mislead the attacker and make him spend a lot of time on the wrong track and, hopefully, give up. To conclude:

- With a different server name in the `Server` header field, you can deflect some automated tools that use this information to find servers of certain make.
- It is possible to fool and confuse a range of attackers with not quite developed skills. Not everyone knows of TCP/IP and HTTP fingerprinting, for example.
- Small changes can be the most effective.

Now, let's see how we can hide server information in practice.

Changing the Server Header Field

The following sections discuss alternative approaches to changing the web server identity.

Changing the name in the source code

You can make modifications to change the web server identity in two places in the source code. One is in the include file *httpd.h* in Apache 1 (*ap_release.h* in Apache 2) where the version macros are defined:

```
#define SERVER_BASEVENDOR   "Apache Group"
#define SERVER_BASEPRODUCT  "Apache"
#define SERVER_BASEREVISION "1.3.29"
#define SERVER_BASEVERSION  SERVER_BASEPRODUCT "/" SERVER_BASEREVISION
#define SERVER_PRODUCT  SERVER_BASEPRODUCT
#define SERVER_REVISION SERVER_BASEREVISION
#define SERVER_VERSION  SERVER_PRODUCT "/" SERVER_REVISION
```

Apache Benchmark recommends that only the value of the `SERVER_BASEPRODUCT` macro be changed, allowing the other information such as the version number to remain in the code so it can be used later, for example, for web server version identification (by way of code audit, not from the outside). If you decide to follow this recommendation, the `ServerTokens` directive must be set to `ProductOnly`, as discussed earlier in this chapter.

The reason Apache Benchmark recommends changing just one macro is because some modules (such as *mod_ssl*) are made to work only with a specific version of the Apache web server. To ensure correct operation, these modules check the Apache version number (contained in the `SERVER_BASEVERSION` macro) and refuse to run if the version number is different from what is expected.

A different approach for changing the name in a source file is to replace the `ap_set_version( )` function, which is responsible for construction of the server name in the first place. For Apache 1, replace the existing function (in *http_main.c*) with one like the following, specifying whatever server name you wish:

```
static void ap_set_version(void)
{
    /* set the server name */
    ap_add_version_component("Microsoft-IIS/5.0");
    /* do not allow other modules to add to it */
    version_locked++;
}
```

For Apache 2, replace the function (defined in *core.c*):

```
static void ap_set_version(apr_pool_t *pconf)
{
    /* set the server name */
    ap_add_version_component(pconf, "Microsoft-IIS/5.0");
    /* do not allow other modules to add to it */
    version_locked++;
}
```

Changing the name using mod_security

Changing the source code can be tiresome, especially if it is done repeatedly. A different approach to changing the name of the server is to use a third-party module, *mod_security* (described in detail in Chapter 12). For this approach to work, we must allow Apache to reveal its full identity, and then instruct *mod_security* to change the identity to something else. The following directives can be added to Apache configuration:

```
# Reveal full identity (standard Apache directive)
ServerTokens Full
# Replace the server name (mod_security directive)
SecServerSignature "Microsoft-IIS/5.0"
```

Apache modules are not allowed to change the name of the server completely, but *mod_security* works by finding where the name is kept in memory and overwriting the text directly. The `ServerTokens` directive must be set to `Full` to ensure the web server allocates a large enough space for the name, giving *mod_security* enough space to make its changes later.

Changing the name using mod_headers with Apache 2

The *mod_headers* module is improved in Apache 2 and can change response headers. In spite of that, you cannot use it to change the two crucial response headers, `Server` and `Date`. But the approach does work when the web server is working in a reverse proxy mode. In that case, you can use the following configuration:

```
Header set Server "Microsoft-IIS/5.0"
```

However, there is one serious problem with this. Though the identity change works in normal conditions, *mod_headers* is not executed in exceptional circumstances. So, for example, if you make an invalid request to the reverse proxy and force it to respond with status code `400` ("Bad request"), the response will include the `Server` header containing the true identity of the reverse proxy server.

Removing Default Content

The key to changing web server identity is consistency. The trouble we went through to change the web server make may be useless if we leave the default Apache content around. The removal of the default content is equivalent to changing one's clothes when going undercover. This action may be useful even if we do not intend to change the server identity. Applications often come with sample programs and, as a general rule, it is a good practice to remove them from production systems; they may contain vulnerabilities that may be exploited later.

Most of the default content is out of reach of the public, since we have built our Apache from scratch, changed the root folder of the web site, and did not include aliases for the manual and the icons. Just to be thorough, erase the following directories:

- */usr/local/apache/cgi-bin*
- */usr/local/apache/htdocs*
- */usr/local/apache/manual* (Apache 2 only)

You will probably want to keep the original */usr/local/apache/logs* directory though the logs are stored in */var/www/logs*. This is because many modules use the *logs/* folder relative to the Apache installation directory to create temporary files. These modules usually offer directives to change the path they use, but some may not. The only remaining bit of default content is the error pages Apache displays when errors occur. These pages can be replaced with the help of the `ErrorDocument` directive. Using one directive per error code, replace the error pages for all HTTP error codes. (A list of HTTP codes is given in Chapter 8; it can also be found at *http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html*.)

```
ErrorDocument 401 /error/401.html
ErrorDocument 403 /error/403.html
ErrorDocument 404 /error/404.html
ErrorDocument 500 /error/500.html
...
```

An alternative to creating dozens of static pages is to create one intelligent script that retrieves the error code from Apache and uses it to display the appropriate message. A small bit of programming is required in this case, following guidance from the Apache documentation at *http://httpd.apache.org/docs-2.0/custom-error.html*.

Putting Apache in Jail

Even the most secure software installations get broken into. Sometimes, this is because you get the attention of a skilled and persistent attacker. Sometimes, a new vulnerability is discovered, and an attacker uses it before the server is patched. Once an intruder gets in, his next step is to look for local vulnerability and become *superuser*. When this happens, the whole system becomes contaminated, and the only solution is to reinstall everything.

Our aim is to contain the intrusion to just a part of the system, and we do this with the help of the `chroot(2)` system call. This system call allows restrictions to be put on a process, limiting its access to the filesystem. It works by choosing a folder to become the new filesystem root. Once the system call is executed, a process cannot go back (in most cases, and provided the jail was properly constructed).

The *root* user can almost always break out of jail. The key to building an escape-proof jail environment is not to allow any *root* processes to exist inside the jail. You must also not have a process outside jail running as the same user as a process inside jail. Under some circumstances, an attacker may jump from one process to another and break out of jail. That's one of the reasons why I have insisted on having a separate account for Apache.

The term *chroot* is often interchangeably used with the term *jail*. The term can be used as a verb and noun. If you say Apache is *chrooted*, for example, you are saying that Apache was put in jail, typically via use of the *chroot* binary or the `chroot(2)` system call. On Linux systems, the meanings of *chroot* and *jail* are close enough. BSD systems have a separate `jail()` call, which implements additional security mechanisms. For more details about the `jail()` call, see the following: *http://docs.freebsd.org/44doc/papers/jail/jail.html*.

Incorporating the jail mechanism (using either `chroot(2)` or `jail()`) into your web server defense gives the following advantages:

Containment
: If the intruder breaks in through the server, he will only be able to access files in the restricted file system. Unable to touch other files, he will be unable to alter them or harm the data in any way.

No shell
: Most exploits need shells (mostly */bin/sh*) to be fully operative. While you cannot remove a shell from the operating system, you can remove it from a jail environment.

Limited tool availability
: Once inside, the intruder will need tools to progress further. To begin with, he will need a shell. If a shell isn't available he will need to find ways to bring one in from the inside. The intruder will also need a compiler. Many black hat tools are not used as binaries. Instead, these tools are uploaded to the server in source and compiled on the spot. Even many automated attack tools compile programs. The best example is the Apache Slapper Worm (see the sidebar "Apache Slapper Worm").

Absence of suid root binaries
: Getting out of a jail is possible if you have the privileges of the *root* user. Since all the effort we put into the construction of a jail would be meaningless if we allowed *suid root* binaries, make sure you do not put such files into the jail.

The `chroot(2)` call was not originally designed as a security measure. Its use for security is essentially a hack, and will be replaced as the server virtualization technologies advance. For Linux, that will happen once these efforts become part of a mainstream kernel. Though server virtualization falls out of the scope of this book, some information on this subject is provided in Chapter 9.

The following sections describe various approaches to putting Apache in jail. First, an example demonstrating use of the original *chroot* binary to put a process in jail is shown. That example demonstrates the issues that typically come up when attempting to put a process in jail and briefly documents tools that are useful for solving these issues. Next, the steps required for creating a jail and putting Apache in it using *chroot* are shown. This is followed by the simpler `chroot(2)` approach, which can be used in some limited situations. Finally, the use of *mod_security* or *mod_chroot* to chroot Apache is presented.

Apache Slapper Worm

The Apache Slapper Worm (*http://www.cert.org/advisories/CA-2002-27.html*) is arguably the worst thing to happen to the Apache web server as far as security goes. It uses vulnerabilities in the OpenSSL subsystem (*http://www.cert.org/advisories/CA-2002-23.html*) to break into a system running Apache. It proceeds to infect other systems and calls back home to become a part of a distributed denial of service (DDoS) network. Some variants install a backdoor, listening on a TCP/IP port. The worm only works on Linux systems running on the Intel architecture.

The behavior of this worm serves as an excellent case study and a good example of how some of the techniques we used to secure Apache help in real life.

- The worm uses a probing request to determine the web server make and version from the `Server` response header and attacks the servers it knows are vulnerable. A fake server signature would, therefore, protect from this worm. Subsequent worm mutations stopped using the probing request, but the initial version did and this still serves as an important point.
- If a vulnerable system is found, the worm source code is uploaded (to */tmp*) and compiled. The worm would not spread to a system without a compiler, to a system where the server is running from a jail, or to a system where code execution in the */tmp* directory is disabled (for example, by mounting the partition with a `noexec` flag).

Proper firewall configuration, as discussed in Chapter 9, would stop the worm from spreading and would prevent the attacker from going into the server through the backdoor.

Tools of the chroot Trade

Before you venture into chroot land you must become aware of several tools and techniques you will need to make things work and to troubleshoot problems when they appear. The general problem you will encounter is that programs do not expect to be run without full access to the filesystem. They assume certain files are present and they do not check error codes of system calls they assume always succeed. As a result, these programs fail without an error message. You must use diagnostic tools such as those described below to find out what has gone wrong.

Sample use of the chroot binary

The *chroot* binary takes a path to the new filesystem root as its first parameter and takes the name of another binary to run in that jail as its second parameter. First, we need to create the folder that will become the jail:

```
# mkdir /chroot
```

Then, we specify the jail (as the *chroot* first parameter) and try (and fail) to run a shell in the jail:

```
# chroot /chroot /bin/bash
chroot: /bin/bash: No such file or directory
```

The above command fails because *chroot* corners itself into the jail as its first action and attempts to run */bin/bash* second. Since the jail contains nothing, *chroot* complains about being unable to find the binary to execute. Copy the shell into the jail and try (and fail) again:

```
# mkdir /chroot/bin
# cp /bin/bash /chroot/bin/bash
# chroot /chroot /bin/bash
chroot: /bin/bash: No such file or directory
```

How can that be when you just copied the shell into jail?

```
# ls -al /chroot/bin/bash
-rwxr-xr-x   1 root    root      605504 Mar 28 14:23 /chroot/bin/bash
```

The *bash* shell is compiled to depend on several shared libraries, and the Linux kernel prints out the same error message whether the problem is that the target file does not exist or that any of the shared libraries it depends on do not exist. To move beyond this problem, we need the tool from the next section.

Using ldd to discover dependencies

The *ldd* tool—available by default on all Unix systems—prints shared library dependencies for a given binary. Most binaries are compiled to depend on shared libraries and will not work without them. Using *ldd* with the name of a binary (or another shared library) as the first parameter gives a list of files that must accompany the binary to work. Trying *ldd* on */bin/bash* gives the following output:

```
# ldd /bin/bash
        libtermcap.so.2 => /lib/libtermcap.so.2 (0x0088a000)
        libdl.so.2 => /lib/libdl.so.2 (0x0060b000)
        libc.so.6 => /lib/tls/libc.so.6 (0x004ac000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00494000)
```

Therefore, *bash* depends on four shared libraries. Create copies of these files in jail:

```
# mkdir /chroot/lib
# cp /lib/libtermcap.so.2 /chroot/lib
# cp /lib/libdl.so.2 /chroot/lib
# cp /lib/tls/libc.so.6 /chroot/lib
# cp /lib/ld-linux.so.2 /chroot/lib
```

The jailed execution of a *bash* shell will finally succeed:

```
# chroot /chroot /bin/bash
bash-2.05b#
```

You are rewarded with a working shell prompt. You will not be able to do much from it though. Though the shell works, none of the binaries you would normally use are available inside (*ls*, for example). You can only use the built-in shell commands, as can be seen in this example:

```
bash-2.05b# pwd
/
bash-2.05b# echo /*
/bin /lib
bash-2.05b# echo /bin/*
/bin/bash
bash-2.05b# echo /lib/*
/lib/ld-linux.so.2 /lib/libc.so.6 /lib/libdl.so.2 /lib/libtermcap.so.2
```

As the previous example demonstrates, from a jailed shell you can access a few files you explicitly copied into the jail and nothing else.

Using strace to see inside processes

The *strace* tool (*truss* on systems other than Linux) intercepts and records system calls that are made by a process. It gives much insight into how programs work, without access to the source code. Using *chroot* and *ldd*, you will be able to get programs to run inside jail, but you will need *strace* to figure out why they fail when they fail without an error message, or if the error message does not indicate the real cause of the problem. For that reason, you will often need *strace* inside the jail itself. (Remember to remove it afterwards.)

Using *strace* you will find that many innocent looking binaries do a lot of work before they start. If you want to experiment, I suggest you write a simple program such as this one:

```
#include <stdio.h>
#include <stdarg.h>

int main(void) {
    puts("Hello world!");
}
```

Compile it once with a shared system support and once without it:

```
# gcc helloworld.c -o helloworld.shared
# gcc helloworld.c -o helloworld.static -static
```

Using *strace* on the static version gives the following output:

```
# strace ./helloworld.static
execve("./helloworld.static", ["./helloworld.static"], [/* 22 vars */]) = 0
uname({sys="Linux", node="ben", ...})   = 0
brk(0)                                  = 0x958b000
brk(0x95ac000)                          = 0x95ac000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xbf51a000
write(1, "Hello world!\n", 13Hello world!
)         = 13
munmap(0xbf51a000, 4096)                = 0
exit_group(13)
```

The *strace* output is ugly. Each line in the output represents a system call made from the process. It is not important at the moment what each line contains. Jailed binaries most often fail because they cannot open a file. If that happens, one of the lines near the end of the output will show the name of the file the binary attempted to access:

```
open("/usr/share/locale/locale.alias", O_RDONLY) = -1 ENOENT
(No such file or directory)
```

As an exercise, use *strace* on the dynamically compiled version of the program and compare the two outputs. You will see how many shared libraries are accessed even from a small program such as this one.

Using chroot to Put Apache in Jail

Now that you know the basics of using chroot to put a process in jail and you are familiar with tools required to facilitate the process, we can take the steps required to put Apache in jail. Start by creating a new home for Apache and move the version installed (shown in the "Installation Instructions" section) to the new location:

```
# mkdir -p /chroot/apache/usr/local
# mv /usr/local/apache /chroot/apache/usr/local
# ln -s /chroot/apache/usr/local/apache /usr/local/apache
# mkdir -p /chroot/apache/var
# mv /var/www /chroot/apache/var/
# ln -s /chroot/apache/var/www /var/www
```

The symbolic link from the old location to the new one allows the web server to be used with or without being jailed as needed and allows for easy web server upgrades.

Like other programs, Apache depends on many shared libraries. The *ldd* tool gives their names (this *ldd* output comes from an Apache that has all default modules built-in statically):

```
# ldd /chroot/apache/usr/local/apache/bin/httpd
        libm.so.6 => /lib/tls/libm.so.6 (0x005e7000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x00623000)
        libgdbm.so.2 => /usr/lib/libgdbm.so.2 (0x00902000)
        libexpat.so.0 => /usr/lib/libexpat.so.0 (0x00930000)
        libdl.so.2 => /lib/libdl.so.2 (0x0060b000)
        libc.so.6 => /lib/tls/libc.so.6 (0x004ac000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00494000)
```

This is a long list; we make copies of these libraries in the jail:

```
# mkdir /chroot/apache/lib
# cp /lib/tls/libm.so.6 /chroot/apache/lib
# cp /lib/libcrypt.so.1 /chroot/apache/lib
# cp /usr/lib/libgdbm.so.2 /chroot/apache/lib
# cp /usr/lib/libexpat.so.0 /chroot/apache/lib
# cp /lib/libdl.so.2 /chroot/apache/lib
# cp /lib/tls/libc.so.6 /chroot/apache/lib
# cp /lib/ld-linux.so.2 /chroot/apache/lib
```

Putting user, group, and name resolution files in jail

Though the *httpd* user exists on the system (you created it as part of the installation earlier); there is nothing about this user in the jail. The jail must contain the basic user authentication facilities:

```
# mkdir /chroot/apache/etc
# cp /etc/nsswitch.conf /chroot/apache/etc/
# cp /lib/libnss_files.so.2 /chroot/apache/lib
```

The jail user database needs to contain at least one user and one group. Use the same name as before and use the identical user and group numbers inside and outside the jail. The filesystem stores user and group numbers to keep track of ownership. It is a job of the *ls* binary to get the usernames from the user list and show them on the screen. If there is one user list on the system and another in the jail with different user numbers, directory listings will not make much sense.

```
# echo "httpd:x:500:500:Apache:/:/sbin/nologin" > /chroot/apache/etc/passwd
# echo "httpd:x:500:" > /chroot/apache/etc/group
```

At this point, Apache is almost ready to run and would run and serve pages happily. A few more files are needed to enable domain name resolution:

```
# cp /lib/libnss_dns.so.2 /chroot/apache/lib
# cp /etc/hosts /chroot/apache/etc
# cp /etc/resolv.conf /chroot/apache/etc
```

Finishing touches for Apache jail preparation

The walls of the jail are now up. Though the following files are not necessary, experience shows that many scripts require them. Add them now to avoid having to debug mysterious problems later.

Construct special devices after using *ls* to examine the existing */dev* folder to learn what numbers should be used:

```
# mkdir /chroot/apache/dev
# mknod -m 666 /chroot/apache/dev/null c 1 3
# mknod -m 666 /chroot/apache/dev/zero c 1 5
# mknod -m 644 /chroot/apache/dev/random c 1 8
```

Then, add a temporary folder:

```
# mkdir /chroot/apache/tmp
# chmod +t /chroot/apache/tmp
# chmod 777 /chroot/apache/tmp
```

Finally, configure the time zone and the locale (we could have copied the whole */usr/share/locale* folder but we will not because of its size):

```
# cp /usr/share/zoneinfo/MET /chroot/apache/etc/localtime
# mkdir -p /chroot/apache/usr/lib/locale
# set | grep LANG
LANG=en_US.UTF-8
LANGVAR=en_US.UTF-8
# cp -dpR /usr/lib/locale/en_US.utf8 /chroot/apache/usr/lib/locale
```

Preparing PHP to work in jail

To make PHP work in jail, you should install it as normal. Establish a list of shared libraries required and copy them into the jail:

```
# ldd /chroot/apache/usr/local/apache/libexec/libphp4.so
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x006ef000)
        libresolv.so.2 => /lib/libresolv.so.2 (0x00b28000)
        libm.so.6 => /lib/tls/libm.so.6 (0x00111000)
        libdl.so.2 => /lib/libdl.so.2 (0x00472000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x00f67000)
        libc.so.6 => /lib/tls/libc.so.6 (0x001df000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00494000)
```

Some of the libraries are already in the jail, so skip them and copy the remaining libraries (shown in bold in the previous output):

```
# cp /lib/libresolv.so.2 /chroot/apache/lib
# cp /lib/libnsl.so.1 /chroot/apache/lib
```

One problem you may encounter with a jailed PHP is that scripts will not be able to send email because the *sendmail* binary is missing. To solve this, change the PHP configuration to make it send email using the SMTP protocol (to localhost or some other SMTP server). Place the following in the *php.ini* configuration file:

```
SMTP = localhost
```

Preparing Perl to work in jail

To make Perl work, copy the files into the jail:

```
# cp -dpR /usr/lib/perl5 /chroot/apache/usr/lib
# mkdir /chroot/apache/bin
# cp /usr/bin/perl /chroot/apache/bin
```

Determine the missing libraries:

```
# ldd /chroot/apache/bin/perl
        libperl.so => /usr/lib/perl5/5.8.1/i386-linux-thread-multi
/CORE/libperl.so (0x0067b000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x00664000)
        libdl.so.2 => /lib/libdl.so.2 (0x0060b000)
        libm.so.6 => /lib/tls/libm.so.6 (0x005e7000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x00623000)
        libutil.so.1 => /lib/libutil.so.1 (0x00868000)
        libpthread.so.0 => /lib/tls/libpthread.so.0 (0x00652000)
        libc.so.6 => /lib/tls/libc.so.6 (0x004ac000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00494000)
```

Then add them to the libraries that are inside:

```
# cp /lib/libutil.so.1 /chroot/apache/lib
# cp /lib/tls/libpthread.so.0 /chroot/apache/lib
```

Taking care of small jail problems

Most CGI scripts send email using the *sendmail* binary. That will not work in our jail since the *sendmail* binary isn't there. Adding the complete *sendmail* installation to the jail would defy the very purpose of having a jail in the first place. If you encounter this problem, consider installing *mini_sendmail* (*http://www.acme.com/software/mini_sendmail/*), a *sendmail* replacement specifically designed for jails. Most programming languages come with libraries that allow email to be sent directly to an SMTP server. PHP can send email directly, and from Perl you can use the *Mail::Sendmail* library. Using these libraries reduces the number of packages that are installed in a jail.

You will probably encounter database connectivity problems when scripts in jail try to connect to a database engine running outside the jail. This happens if the program is using *localhost* as the host name of the database server. When a database client library sees *localhost,* it tries to connect to the database using a Unix domain socket. This socket is a special file usually located in */tmp*, */var/run*, or */var/lib*, all outside the jail. One way to get around this is to use `127.0.0.1` as the host name and force the database client library to use TCP/IP. However, since a performance penalty is involved with that solution (Unix domain socket communication is much faster than communication over TCP/IP), a better way would be to have the socket file in the jail.

For PostgreSQL, find the file *postgresql.conf* (usually in */var/lib/pgsql/data*) and change the line containing the `unix_socket_directory` directive to read:

```
unix_socket_directory = '/chroot/apache/tmp'
```

Create a symbolic link from the previous location to the new one:

```
# ln -s /chroot/apache/tmp/.s.PGSQL.5432 /tmp
```

MySQL keeps its configuration options in a file called *my.cnf*, usually located in */etc*. In the same file, you can add a client section (if one is not there already) and tell clients where to look for a socket:

```
[mysqld]
datadir=/var/lib/mysql
socket=/chroot/apache/var/lib/mysql/mysql.sock

[client]
socket=/chroot/apache/var/lib/mysql/mysql.sock
```

Or, just as you did with PostgreSQL, create a symbolic link:

```
# mkdir -p /chroot/apache/var/lib/mysql
# chown mysql /chroot/apache/var/lib/mysql/
# ln -s /chroot/apache/var/lib/mysql/mysql.sock /var/lib/mysql
```

Using the chroot(2) Patch

Now that I have explained the manual chroot process, you are wondering if an easier way exists. The answer is, conditionally, yes.

The approach so far was to create the jail before the main process was started. For this approach to work, the jail must contain all shared libraries and files the process requires. This approach is also known as an *external chroot*.

With an *internal chroot*, the jail is established from within the process after the process initialization is completed. In the case of Apache, the jail must be created before request processing begins, at the latest. The process is born free and then jailed. Since the process has full access to the filesystem during the initialization phase, it is free to access any files it needs. Because of the way chrooting works, descriptors to the files opened before the call remain valid after. Therefore, we do not have to create a copy of the filesystem and we can have a "perfect" jail, the one that contains only files needed for web serving, the files in the web server tree.

Internal chroot can be dangerous. In external chroot approaches, the process is born in jail, so it has no opportunity to interact with the outside filesystem. With the internal chroot, however, the process has full access to the filesystem in the beginning and this allows it to open files outside the jail and continue to use them even after the jail is created. This opens up interesting opportunities, such as being able to keep the logs and the binaries outside jail, but is a potential problem. Some people are not comfortable with leaving open file descriptors outside jail. You can use the *lsof* utility to see which file descriptors Apache has open and determine whether any of them point outside jail. My recommendation is the following: If you can justify a high level of security for your installation, go for a proper external chroot approach. For installations of less importance, spending all that time is not feasible. In such cases, use the internal chroot approach.

It is obvious that internal chrooting is not a universal solution. It works only if the following is true:

- The only functionality needed is that of Apache and its modules.
- There will be no processes (such as CGI scripts) started at runtime. Alternatively, if CGI scripts are used, they will be statically compiled.
- Access to files outside the web server root will be not be required at runtime. (For example, if you intend to use the piped logging mechanism, Apache must be able to access the logging binary at runtime to restart logging in case the original logging process dies for some reason. Piped logging is discussed in Chapter 8.)

Now that I have lured you into thinking you can get away from the hard labor of chrooting, I will have to disappoint you: Apache does not support internal chrooting natively. But the help comes from Arjan de Vet in the form of a `chroot(2)` patch. It is available for download from *http://www.devet.org/apache/chroot/*. After the patch is applied to the source code, Apache will support a new directive, `ChrootDir`. Chrooting Apache can be as easy as supplying the new root of the filesystem as the `ChrootDir` first parameter. The record of a successful `chroot(2)` call will be in the error log.

As a downside, you will have to apply the patch every time you install Apache. And there is the problem of finding the patch for the version of Apache you want to install. At the time of this writing only the patch for Apache 1.3.31 is available. But not everything is lost.

Using mod_security or mod_chroot

In a saga with more twists than a soap opera, I will describe a third way to jail Apache. Provided the limitations described in the previous section are acceptable to you, this method is the simplest: chrooting using *mod_security* (*http://www.modsecurity.org*) or *mod_chroot* (*http://core.segfault.pl/~hobbit/mod_chroot/*). Both modules use the same method to do their work (at the time of this writing) so I will cover them in this section together. Which module you will use depends on your circumstances. Use *mod_security* if you have a need for its other features. Otherwise, *mod_chroot* is likely to be a better choice because it only contains code to deal with this one feature and is, therefore, less likely to have a fault.

The method these two modules use to perform chrooting is a variation of the `chroot(2)` patch. Thus, the discussion about the usefulness of the `chroot(2)` patch applies to this case. The difference is that here the `chroot(2)` call is made from within the Apache module (*mod_security* or *mod_chroot*), avoiding a need to patch the Apache source code. And it works for 1.x and 2.x branches of the server. As in the previous case, there is only one new directive to learn: `SecChrootDir` for *mod_security* or `ChrootDir` for *mod_chroot*. Their syntaxes are the same, and they accept the name of the root directory as the only parameter:

```
SecChrootDir /chroot/apache
```

The drawback of working from inside the module is that it is not possible to control exactly when the chroot call will be executed. But, as it turns out, it is possible to successfully perform a `chroot(2)` call if the module is configured to initialize last.

Apache 1

For Apache 1, this means manually configuring the module loading order to make sure the chroot module initializes last. To find a list of compiled-in modules, execute the *httpd* binary with the `-l` switch:

```
# ./httpd -l
Compiled-in modules:
  http_core.c
  mod_env.c
  mod_log_config.c
```

```
    mod_mime.c
    mod_negotiation.c
    mod_status.c
    mod_include.c
    mod_autoindex.c
    mod_dir.c
    mod_cgi.c
    mod_asis.c
    mod_imap.c
    mod_actions.c
    mod_userdir.c
    mod_alias.c
    mod_rewrite.c
    mod_access.c
    mod_auth.c
    mod_so.c
    mod_setenvif.c
```

To this list, add modules you want to load dynamically. The core module, *http_core*, should not appear on your list. Modules will be loaded in the reverse order from the one in which they are listed in the configuration file, so *mod_security* (or *mod_chroot*) should be the first on the list:

```
ClearModuleList
AddModule mod_security.c
AddModule ...
AddModule ...
```

Apache 2

With Apache 2, there is no need to fiddle with the order of modules since the new API allows module programmers to choose module position in advance. However, the changes in the architecture are causing other potential problems to appear:

- Unlike in Apache 1, in Apache 2 some of the initialization happens after the last module initializes. This causes problems if you attempt to create a jail in which the logs directory stays outside jail. The solution is to create another logs directory inside jail, which will be used to store the files Apache 2 needs (e.g., the *pid* file). Many of the modules that create temporary files have configuration directives that change the paths to those files, so you can use those directives to have temporary files created somewhere else (but still within the jail).
- On some platforms, internal Apache 2 chroot does not work if the `AcceptMutex` directive is set to `pthread`. If you encounter a problem related to mutexes change the setting to something else (e.g., `posixsem`, `fcntl`, or `flock`).