

## Securing Apache 2: Step-by-Step

Artur Maj 2004-06-21

When choosing a web server, Apache [very often](#) wins against its competitors because of stability, performance, that fact that it's open source, and many other advantages. But when deciding on which version of Apache to use, the choice is not always so simple. On the one hand there is a very popular, stable version used by millions of users, version 1.3, and on the other hand, there is an enhanced and re-designed version 2.0.

And even if the new version has got a lot more extensions and features, some people still decide to use version 1.3, because in their opinion this branch is more stable and secure. As a matter of fact, there is some truth in this statement. Since version 1.3 has been used by millions of users for a long time, most security holes in this version are very likely to be already discovered. At the same time version 2.0 may have many more as-yet undiscovered vulnerabilities, just sleeping and waiting to be found.

Continuing the step-by-step fashion from the previous series ([Securing Apache](#), [Securing PHP](#), and [Securing MySQL](#)), this article shows how to install and configure Apache 2.0 to minimize the risk of unauthorized access or successful break-in, even if new security vulnerabilities in Apache web server are found. Thus, it will be possible to enjoy the new features of Apache 2.0 without worrying too much about its security bugs, regardless if they are only imaginary, or are in fact real and serious threats.

### Functionality requirements

In the world of security, there are a few golden principles that should always be followed. One such principle is the rule which says that only absolutely required parts of the software should be used. All other components should be disabled, made inaccessible or not even be installed at all.

The logic behind this rule is very simple -- if there is software with dozens of components that are enabled by default, finding only one security vulnerability in any one of these components can put the whole system at risk of a successful break-in. On the other hand, if only a few absolutely necessary components are enabled, finding a new security bug doesn't necessary mean that the software is vulnerable -- because the discovered bug may affect components that are not enabled, or are not installed. The probability of a successful break-in in this case is obviously much lower than in case of the default installation.

Therefore, before starting to secure Apache 2, it is very important to know what functionality we really expect from the web server. This will allow us to prepare the list of modules that we will leave enabled, while the rest will be disabled during compilation time.

According to this rule, this article assumes that very basic functionality of Apache will be used:

- Only static HTML pages will be served.
- The server must support the virtual hosting mechanism.
- Access to some web pages will be restricted to selected IP addresses or users (basic authentication).

- The server must log all web requests (including information about web browsers).

One can note that the above functionality doesn't support CGI scripts, the SSL protocol or other useful Apache features. This is because the main purpose of the article is to present a general method of securing Apache 2.0, without focusing on a particular implementation. If there is a need for additional functionality, readers can still use the presented solution as a starting point, and enhance it by enabling additional modules, for example, *mod\_ssl*, *mod\_cgi* or others.

## Security assumptions

To provide as many security layers as possible, and at the same time keep this solution portable among many different Linux/BSD systems, the following layers of security will be used:

### the network environment

- The web server should be protected by a firewall; the rules should accept incoming requests to port 80/tcp and allow outgoing HTTP responses. Except for certain ICMP messages (e.g. *source-quench*, *time-exceed*, *parameter-problem*, *destination-unreachable*), all other packets should be dropped or denied.
- An intrusion detection (or prevention) system should be used; Apache's log files should also be monitored.

### the operating system

- The operating system should be hardened as much as possible; all unnecessary components should be removed from the system.
- If supported, the operating system should not allow executing programs on the stack.
- All unnecessary network services should be disabled.
- The number of SUID/SGID files should be minimized.

### the Apache web server

- Only absolutely necessary Apache modules should be enabled; the rest should be disabled during compilation time.
- All diagnostic web pages and the automatic directory indexing service must be turned off.
- The server should disclose the least amount of information about itself as possible -- security through obscurity. Although this is not a real security layer, applying it will at least make the attacks a little bit more difficult to perform.
- The web server must run under a dedicated UID/GID, not one used by any other system process.
- Apache's processes must have limited access to the file systems (chrooting).
- In the Apache chrooted environment there cannot be any shell program present (*/bin/sh*, */bin/csh* etc.) -- it makes the process of executing exploits much more difficult to perform.

## Installing the operating system

First and foremost, we must choose an operating system upon which the web server will run. The rest of article presents how to secure Apache on FreeBSD (5.1), however readers are free to use their favorite Unix, BSD, Linux or Linux-like operating system.

With regards to our security assumptions, after installing the operating system it must be hardened against both remote and local attacks. Regardless of the chosen UNIX/Linux/BSD distribution, it is very important to install only the core operating system, remove any redundant packages and apply up-to-date patches to the kernel and all installed software.

It is also recommended to periodically synchronize the local clock against a trusted time server, using the Network Time Protocol (NTP), and to send log files to a remote, dedicated log server.

After the system is prepared, we can start installing Apache 2.0. The first step is to add a new group and regular user called *apache*. An example from FreeBSD has been shown below:

```
pw groupadd apache
pw useradd apache -c "Apache Server" -d /dev/null -g apache -s /sbin/nologin
```

The Apache child processes will run with the privileges of the group and user *apache*. Since the above account will be dedicated to the Apache web server, this will provide separation of privileges and avoid potential security problems when several different processes are being run under the same account, e.g. user *nobody*.

## Downloading the software

Next, the latest version of Apache 2.0 software should be downloaded from the [Apache website](#), and then unpacked. Since we want to disable unnecessary modules during compilation time, it is very important to download the source code, not binaries. It is also important to test the downloaded software against a PGP signature, to make sure that the downloaded version is complete and unmodified.

```
lynx http://httpd.apache.org/download.cgi
  <download: httpd-2.0.xx.tar.gz, httpd-2.0.xx.tar.gz.asc, KEYS>
gpg --import KEYS
gpg httpd-2.0.49.tar.gz.asc
  gpg: Good signature from "Sander Striker <striker@apache.org>"
tar zxvf httpd-2.0.49.tar.gz
cd ./httpd-2.0.49/
```

## Choosing Apache's modules

After the Apache source code is unpacked, we must choose which modules will remain enabled, and which will

be removed. A short description of all modules available in Apache 2.0 can be found at <http://httpd.apache.org/docs-2.0/mod/>.

To fulfill the functionality and security requirements assumed at the beginning of this article, we will compile only the following modules:

Module's name	Description
core	The core Apache features, required in every Apache installation.
http_core	The core http support, required in every Apache 2.0 installation.
prefork	Multi-Processing Module (MPM) that implements a non-threaded, pre-forking web server. Can be replaced by other multiprocessing module, e.g. <i>worker</i> , <i>threadpool</i> etc. The MPM module is required in every Apache 2.0 installation.
mod_access	Provides access control based on client hostname, IP address, or other characteristics of the client request. Because this module is needed to use "order", "allow" and "deny" directives, it should remain enabled.
mod_auth	Required in order to implement user authentication using text files (HTTP Basic Authentication), which was specified in functionality assumptions.
mod_dir	Required to search and serve directory index files: "index.html", "default.htm", etc.
mod_log_config	Required to implement logging of the requests made to the server.
mod_mime	Required to set the character set, content-encoding, handler, content-language, and MIME types of documents.

Since we want to enable only the minimal number of modules, we will compile all the modules statically. Thanks to that, we will eliminate possibility of occurring vulnerabilities in one more module -- *mod\_so*.

## Compiling and installing the software

In this step we will configure, compile, and install the Apache web server as follows:

```
./configure \  
--prefix=/usr/local/apache2 \  
--with-mpm=prefork \  
--disable-charset-lite \  
--disable-include \  
--disable-env \  
--disable-setenvif \  
--disable-status \  
--disable-autoindex \  
--disable-asis \  
--disable-cgi \  
--disable-negotiation \  
--disable-imap \  
--disable-actions \  
--disable-userdir \  
--disable-alias \  
--disable-so  
make  
su  
umask 022  
make install  
chown -R root:sys /usr/local/  
apache2
```

After Apache is installed, we should make sure that only the following modules are enabled:

```
/usr/local/apache2/bin/httpd -  
l  
Compiled in modules:  
  core.c  
  mod_access.c  
  mod_auth.c  
  mod_log_config.c  
  prefork.c  
  http_core.c  
  mod_mime.c  
  mod_dir.c
```

## Configuring Apache

Before running Apache for the first time, we also need to modify the Apache configuration file. We need to do

this because the default configuration file uses modules that we disabled, and without modifications Apache will not run.

Thus, we must remove the `/usr/local/apache2/conf/httpd.conf` file and create a new `httpd.conf` in its place, with the following content:

```
# =====
# Basic settings
# =====
Listen 0.0.0.0:80
User apache
Group apache
ServerAdmin webmaster@www.ebank.lab
UseCanonicalName Off
ServerSignature Off
HostnameLookups Off
ServerTokens Prod
ServerRoot "/usr/local/apache2"
DocumentRoot "/www"
PidFile /usr/local/apache2/logs/httpd.pid
ScoreBoardFile /usr/local/apache2/logs/httpd.scoreboard
<IfModule mod_dir.c>
    DirectoryIndex index.html
</IfModule>

# =====
# HTTP and performance settings
# =====
Timeout 300
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 15
<IfModule prefork.c>
    MinSpareServers 5
    MaxSpareServers 10
    StartServers 5
    MaxClients 150
    MaxRequestsPerChild 0
</IfModule>

# =====
# Access control
# =====
```

```
<Directory />
Options None
    AllowOverride None
    Order deny,allow
    Deny from all
</Directory>
<Directory "/www/www.ebank.lab">
    Order allow,deny
    Allow from all
</Directory>
<Directory "/www/www.test.lab">
    Order allow,deny
    Allow from all
</Directory>

# =====
# MIME encoding
# =====
<IfModule mod_mime.c>
    TypesConfig /usr/local/apache2/conf/mime.types
</IfModule>
DefaultType text/plain
<IfModule mod_mime.c>
    AddEncoding x-compress .Z
    AddEncoding x-gzip .gz .tgz
    AddType application/x-compress .Z
    AddType application/x-gzip .gz .tgz
    AddType application/x-tar .tgz
</IfModule>

# =====
# Logs
# =====
LogLevel warn
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\""
combined
LogFormat "%h %l %u %t \"%r\" %>s %b" common
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{User-agent}i" agent
ErrorLog /usr/local/apache2/logs/error_log
CustomLog /usr/local/apache2/logs/access_log combined

# =====
```

```
# Virtual hosts
# =====
NameVirtualHost *
<VirtualHost *>
    DocumentRoot "/www/www.ebank.lab"
    ServerName "www.ebank.lab"
    ServerAlias "www.e-bank.lab"
    ErrorLog logs/www.ebank.lab/error_log
    CustomLog logs/www.ebank.lab/access_log combined
</VirtualHost>
<VirtualHost *>
    DocumentRoot "/www/www.test.lab"
    ServerName "www.test.lab"
    ErrorLog logs/www.test.lab/error_log
    CustomLog logs/www.test.lab/access_log combined
</VirtualHost>
```

Compared to the default configuration file, the following important changes have been made:

- The number of enabled modules has been reduced to minimum.
- Apache's processes (except for the *root* process) are set to be executed with unique regular user/group privileges.
- Apache discloses the least information about itself as possible.
- Access rights to the website's content are set to be more restrictive.

According to our functionality requirements, the above configuration assumes that there are two virtual hosts supported by Apache:

- *www.ebank.lab* (alias: *www.e-bank.lab*)
- *www.test.lab*

The content of the above virtual hosts will be physically kept under the */www* directory, so before running Apache we also need to create the corresponding directories with sample web pages:

```
mkdir -p /www/www.ebank.lab
mkdir -p /www/www.test.lab
echo "<html><head><title>eBank.lab</title></head><body>eBank.
lab
  works!</body></html>" > /www/www.ebank.lab/index.html
echo "<html><head><title>test.lab</title></head><body>Test.lab
  works!</body></html>" > /www/www.test.lab/index.html
chmod -R 755 /www
chown -R root:sys /www
```

We must also prepare directories for storing our log files:

```
mkdir -p /usr/local/apache2/logs/www.ebank.
lab
mkdir -p /usr/local/apache2/logs/www.test.lab
chmod -R 755 /usr/local/apache2/logs
chown -R root:sys /usr/local/apache2/logs
```

Finally, we can try to run Apache, and test if everything works properly:

```
/usr/local/apache2/bin/apachectl start
```

If the *www.ebank.lab* website is accessible from a web browser, we can shutdown Apache:

```
/usr/local/apache2/bin/apachectl
stop
```

and then proceed to chroot the server. If there are problems, log files should be analyzed, or the *truss* command (for BSD and Solaris users) should be used, as follows:

```
truss /usr/local/apache2/bin/
httpd
```

Note that for Linux users, the equivalent command is *strace*. Either way, analyzing the output of the *truss* (or *strace*) command should help with finding the reason of failure.

## Chrooting the server

The next step is to limit the Apache processes' access to the filesystems. The chrooting technique was described in detail in the [previous article](#), so at this point we will simply create a directory structure for our new Apache:

```
mkdir -p /chroot/httpd/dev
mkdir -p /chroot/httpd/etc
mkdir -p /chroot/httpd/var/run
mkdir -p /chroot/httpd/usr/lib
mkdir -p /chroot/httpd/usr/libexec
mkdir -p /chroot/httpd/usr/local/apache2/bin
mkdir -p /chroot/httpd/usr/local/apache2/lib
mkdir -p /chroot/httpd/usr/local/apache2/logs/www.ebank.
lab
mkdir -p /chroot/httpd/usr/local/apache2/logs/www.test.lab
mkdir -p /chroot/httpd/usr/local/apache2/conf
mkdir -p /chroot/httpd/usr/local/lib
mkdir -p /chroot/httpd/www
```

The owner of all the above directories should be *root*, and access rights should not allow regular users to perform any changes in these directories:

```
chown -R root:sys /chroot/
httpd
chmod -R 0755 /chroot/httpd
```

Next, we will create the special device file, */dev/null*:

```
ls -al /dev/null
crw-rw-rw- 1 root wheel 2, 2 Mar 14 12:53 /dev/
null
mknod /chroot/httpd/dev/null c 2 2
chown root:sys /chroot/httpd/dev/null
chmod 666 /chroot/httpd/dev/null
```

We also need to create a */chroot/httpd/dev/log* device that is needed for the server to work properly. In the case of our FreeBSD system, the following line should be added to */etc/rc.conf* :

```
syslogd_flags="-l /chroot/httpd/dev/
log"
```

In order for the changes to take effect, we also need to restart the syslogd daemon with the new parameter:

```
kill `cat /var/run/syslog.pid`  
/usr/sbin/syslogd -ss -l /chroot/httpd/dev/  
log
```

The next step is to copy all necessary programs, libraries and configuration files into the new directory tree. In the case of FreeBSD 5.1 the list of required files is as follows:

```
cp /usr/local/apache2/bin/httpd /chroot/httpd/usr/local/apache2/bin/  
cp /usr/local/apache2/lib/libaprutil-0.so.9 /chroot/httpd/usr/local/apache2/  
lib/  
cp /usr/local/apache2/lib/libapr-0.so.9 /chroot/httpd/usr/local/apache2/lib/  
cp /usr/local/apache2/conf/mime.types /chroot/httpd/usr/local/apache2/conf/  
cp /usr/local/apache2/conf/httpd.conf /chroot/httpd/usr/local/apache2/conf/  
cp /usr/local/lib/libexpat.so.4 /chroot/httpd/usr/local/lib/  
cp /usr/lib/libc.so.5 /chroot/httpd/usr/lib/  
cp /usr/lib/libcrypt.so.2 /chroot/httpd/usr/lib/  
cp /usr/lib/libm.so.2 /chroot/httpd/usr/lib/  
cp /usr/libexec/ld-elf.so.1 /chroot/httpd/usr/libexec/  
cp /var/run/ld-elf.so.hints /chroot/httpd/var/run/  
cp /etc/hosts /chroot/httpd/etc/  
cp /etc/nsswitch.conf /chroot/httpd/etc/  
cp /etc/resolv.conf /chroot/httpd/etc/  
cp /etc/group /chroot/httpd/etc/  
cp /etc/master.passwd /chroot/httpd/etc/passwords
```

In the case of other Unix, BSD, Linux and Linux-like systems, the list of required files can be determined by using commands like *ldd*, *strace*, *truss* or *strings*, as was described in the [previous article](#).

After the above steps are done, we need to prepare the password database that must be present in the chrooted filesystem. Thus, from */chroot/httpd/etc/passwords* and */chroot/httpd/etc/group* we have to remove all the lines except *apache*. Next, we should build the password database as follows:

```
cd /chroot/httpd/etc  
pwd_mkdb -d /chroot/httpd/etc  
passwords  
rm -rf /chroot/httpd/etc/master.passwd
```

The above commands should be executed when using FreeBSD. In other systems it may be sufficient to edit the `/chroot/httpd/etc/passwd` and `/chroot/httpd/etc/shadow` files.

Finally, we can copy the sample website content to the chrooted environment:

```
cp -R /www/* /chroot/httpd/  
www/
```

and test if the Apache web server runs correctly:

```
chroot /chroot/httpd /usr/local/apache2/bin/  
httpd
```

## Final steps

If your Apache now works properly, the only thing that is left to do is to create a script that will start Apache during system boot. In order to do this, the [apache.sh](#) script can be used, with the following content:

```
#!/bin/sh  
CHROOT=/chroot/httpd  
HTTPD=/usr/local/apache2/bin/httpd  
PIDFILE=/usr/local/apache2/logs/httpd.pid  
  
echo -n " apache"  
  
case "$1" in  
start)  
    /usr/sbin/chroot $CHROOT $HTTPD  
    ;;  
stop)  
    kill `cat ${CHROOT}/${PIDFILE}`  
    ;;  
*)  
    echo ""  
    echo "Usage: `basename $0` {start|stop}"  
>&2  
    exit 64  
    ;;  
esac  
  
exit 0
```

The above script should be copied to the directory where by default startup scripts are held. In the case of FreeBSD it is the `/usr/local/etc/rc.d` directory. The access rights to that file should be set as follows:

```
chown root:sys /usr/local/etc/rc.d/apache.  
sh  
chmod 711 /usr/local/etc/rc.d/apache.sh
```

## Summary

The main goal of this article was to present a method of securing Apache 2.0 that lets readers mitigate the risk of a successful break-in, even if new vulnerabilities in this software are found. It has been shown how to install Apache with a minimal number of modules, how to set up a more restrictive configuration, and how to implement protection against a large number of exploits by running the web server in a chrooted environment, without the use of any shell programs. And although no method can assure a 100% security, applying the above recommendations will at least make performing a web attack against Apache 2.0 much more difficult, as compared to the default installation.

### Relevant links

Securing Apache: Step-by-Step: <http://www.securityfocus.com/infocus/1694>

Securing PHP: Step-by-Step: <http://www.securityfocus.com/infocus/1706>

Securing MySQL: Step-by-Step: <http://www.securityfocus.com/infocus/1726>

Apache HTTP Server Project: <http://httpd.apache.org>

Sample httpd.conf: [httpd.conf](#)

Sample apache.sh: [apache.sh](#)

### About the author

[Artur Maj](#) works as a Principal Software Engineer for Oracle Corporation, in the EMEA Mobile, Wireless & Voice Center of Expertise. He is experienced in designing computer systems, performing security audits as well as providing security training. He is also author of many articles and publications devoted to securing computer systems and software against intruders.

View [all articles](#) by Artur Maj on SecurityFocus.

*Comments or reprint requests can be sent to the [editor](#).*

[Privacy Statement](#)

Copyright 2006, SecurityFocus