Security Tips

Available Languages: en | es | fr | ko | tr

Some hints and tips on security issues in setting up a web server. Some of the suggestions will be general, others specific to Apache.

- Keep up to Date
- Denial of Service (DoS) attacks
- Permissions on ServerRoot Directories
- Server Side Includes
- CGI in General
- Non Script Aliased CGI
- Script Aliased CGI
- Other sources of dynamic content
- Dynamic content security
- Protecting System Settings
- Protect Server Files by Default
- Watching Your Logs
- Merging of configuration sections

See also

- Comments

Keep up to Date

The Apache HTTP Server has a good record for security and a developer community highly concerned about security issues. But it is inevitable that some problems -- small or large -- will be discovered in software after it is released. For this reason, it is crucial to keep aware of updates to the software. If you have obtained your version of the HTTP Server directly from Apache, we highly recommend you subscribe to the Apache HTTP Server Announcements List where you can keep informed of new releases and security updates. Similar services are available from most third-party distributors of Apache software.

Of course, most times that a web server is compromised, it is not because of problems in the HTTP Server code. Rather, it comes from problems in add-on code, CGI scripts, or the underlying Operating System. You must therefore stay aware of problems and updates with all the software on your system.

Denial of Service (DoS) attacks

All network servers can be subject to denial of service attacks that attempt to prevent responses to clients by tying up the resources of the server. It is not possible to prevent such attacks entirely, but you can do certain things to mitigate the problems that they create.

Often the most effective anti-DoS tool will be a firewall or other operating-system configurations. For example, most firewalls can be configured to restrict the number of simultaneous connections from any individual IP address or network, thus preventing a range of simple attacks. Of course this is no help against Distributed Denial of Service attacks (DDoS).

There are also certain Apache HTTP Server configuration settings that can help mitigate problems:

- The `RequestReadTimeout` directive allows to limit the time a client may take to send the request.
- The `TimeOut` directive should be lowered on sites that are subject to DoS attacks. Setting this to as low as a few seconds may be appropriate. As `TimeOut` is currently used for several different operations, setting it to a low value introduces problems with long running CGI scripts.
- The `KeepAliveTimeout` directive may be also lowered on sites that are subject to DoS attacks. Some sites even turn off the keepalives completely via `KeepAlive`, which has of course other drawbacks on performance.
- The values of various timeout-related directives provided by other modules should be checked.
- The directives `LimitRequestBody`, `LimitRequestFields`, `LimitRequestFieldSize`, `LimitRequestLine`, and `LimitXMLRequestBody` should be carefully configured to limit resource consumption triggered by client input.
- On operating systems that support it, make sure that you use the `AcceptFilter` directive to offload part of the request processing to the operating system. This is active by default in Apache httpd, but may require reconfiguration of your kernel.
- Tune the `MaxRequestWorkers` directive to allow the server to handle the maximum number of simultaneous connections without running out of resources. See also the performance tuning documentation.

- The use of a threaded mpm may allow you to handle more simultaneous connections, thereby mitigating DoS attacks. Further, the `event` mpm uses asynchronous processing to avoid devoting a thread to each connection. Due to the nature of the OpenSSL library the `event` mpm is currently incompatible with `mod_ssl` and other input filters. In these cases it falls back to the behaviour of the `worker` mpm.
- There are a number of third-party modules available through http://modules.apache.org/ that can restrict certain client behaviors and thereby mitigate DoS problems.

Permissions on ServerRoot Directories

In typical operation, Apache is started by the root user, and it switches to the user defined by the `User` directive to serve hits. As is the case with any command that root executes, you must take care that it is protected from modification by non-root users. Not only must the files themselves be writeable only by root, but so must the directories, and parents of all directories. For example, if you choose to place ServerRoot in `/usr/local/apache` then it is suggested that you create that directory as root, with commands like these:

```
mkdir /usr/local/apache
cd /usr/local/apache
mkdir bin conf logs
chown 0 . bin conf logs
chgrp 0 . bin conf logs
chmod 755 . bin conf logs
```

It is assumed that `/`, `/usr`, and `/usr/local` are only modifiable by root. When you install the `httpd` executable, you should ensure that it is similarly protected:

```
cp httpd /usr/local/apache/bin
chown 0 /usr/local/apache/bin/httpd
chgrp 0 /usr/local/apache/bin/httpd
chmod 511 /usr/local/apache/bin/httpd
```

You can create an htdocs subdirectory which is modifiable by other users -- since root never executes any files out of there, and shouldn't be creating files in there.

If you allow non-root users to modify any files that root either executes or writes on then you open your system to root compromises. For example, someone could replace the `httpd` binary so that the next time you start it, it will execute some arbitrary code. If the logs directory is writeable (by a non-root user), someone could replace a log file with a symlink to some other system file, and then root might overwrite that file with arbitrary data. If the log files themselves are writeable (by a non-root user), then someone may be able to overwrite the log itself with bogus data.

Server Side Includes

Server Side Includes (SSI) present a server administrator with several potential security risks.

The first risk is the increased load on the server. All SSI-enabled files have to be parsed by Apache, whether or not there are any SSI directives included within the files. While this load increase is minor, in a shared server environment it can become significant.

SSI files also pose the same risks that are associated with CGI scripts in general. Using the `exec cmd` element, SSI-enabled files can execute any CGI script or program under the permissions of the user and group Apache runs as, as configured in `httpd.conf`.

There are ways to enhance the security of SSI files while still taking advantage of the benefits they provide.

To isolate the damage a wayward SSI file can cause, a server administrator can enable suexec as described in the CGI in General section.

Enabling SSI for files with `.html` or `.htm` extensions can be dangerous. This is especially true in a shared, or high traffic, server environment. SSI-enabled files should have a separate extension, such as the conventional `.shtml`. This helps keep server load at a minimum and allows for easier management of risk.

Another solution is to disable the ability to run scripts and programs from SSI pages. To do this replace `Includes` with `IncludesNOEXEC` in the `Options` directive. Note that users may still use `<--#include virtual="..." -->` to execute CGI scripts if these scripts are in directories designated by a `ScriptAlias` directive.

CGI in General

First of all, you always have to remember that you must trust the writers of the CGI scripts/programs or your ability to spot potential security holes in CGI, whether they were deliberate or accidental. CGI scripts can run essentially arbitrary commands on your system with the permissions of the web server user and can therefore be extremely dangerous if they are not carefully checked.

All the CGI scripts will run as the same user, so they have potential to conflict (accidentally or deliberately) with other scripts e.g. User A hates User B, so he writes a script to trash User B's CGI database. One program which can be used to allow scripts to run as different users is suEXEC which is included with Apache as of 1.2 and is called from special hooks in the Apache server code. Another popular way of doing this is with CGIWrap.

Non Script Aliased CGI

Allowing users to execute CGI scripts in any directory should only be considered if:

- You trust your users not to write scripts which will deliberately or accidentally expose your system to an attack.
- You consider security at your site to be so feeble in other areas, as to make one more potential hole irrelevant.
- You have no users, and nobody ever visits your server.

Script Aliased CGI

Limiting CGI to special directories gives the admin control over what goes into those directories. This is inevitably more secure than non script aliased CGI, but only if users with write access to the directories are trusted or the admin is willing to test each new CGI script/program for potential security holes.

Most sites choose this option over the non script aliased CGI approach.

Other sources of dynamic content

Embedded scripting options which run as part of the server itself, such as `mod_php`, `mod_perl`, `mod_tcl`, and `mod_python`, run under the identity of the server itself (see the `User` directive), and therefore scripts executed by these engines potentially can access anything the server user can. Some scripting engines may provide restrictions, but it is better to be safe and assume not.

Dynamic content security

When setting up dynamic content, such as mod_php, mod_perl or mod_python, many security considerations get out of the scope of httpd itself, and you need to consult documentation from those modules. For example, PHP lets you setup Safe Mode, which is most usually disabled by default. Another example is Suhosin, a PHP addon for more security. For more information about those, consult each project documentation.

At the Apache level, a module named mod_security can be seen as a HTTP firewall and, provided you configure it finely enough, can help you enhance your dynamic content security.

Protecting System Settings

To run a really tight ship, you'll want to stop users from setting up .htaccess files which can override security features you've configured. Here's one way to do it.

In the server configuration file, put

```
<Directory "/">
    AllowOverride None
</Directory>
```

This prevents the use of .htaccess files in all directories apart from those specifically enabled.

Note that this setting is the default since Apache 2.3.9.

Protect Server Files by Default

One aspect of Apache which is occasionally misunderstood is the feature of default access. That is, unless you take steps to change it, if the server can find its way to a file through normal URL mapping rules, it can serve it to clients.

For instance, consider the following example:

```
# cd /; ln -s / public_html
Accessing http://localhost/~root/
```

This would allow clients to walk through the entire filesystem. To work around this, add the following block to your server's configuration:

```
<Directory "/">
    Require all denied
</Directory>
```

This will forbid default access to filesystem locations. Add appropriate Directory blocks to allow access only in those areas you wish. For example,

```
<Directory "/usr/users/*/public_html">
    Require all granted
</Directory>
<Directory "/usr/local/httpd">
    Require all granted
</Directory>
```

Pay particular attention to the interactions of Location and Directory directives; for instance, even if <Directory "/"> denies access, a <Location "/"> directive might overturn it.

Also be wary of playing games with the UserDir directive; setting it to something like ./ would have the same effect, for root, as the first example above. We strongly recommend that you include the following line in your server configuration files:

```
UserDir disabled root
```

Watching Your Logs

To keep up-to-date with what is actually going on against your server you have to check the Log Files. Even though the log files only reports what has already happened, they will give you some understanding of what attacks is thrown against the server and allow you to check if the necessary level of security is present.

A couple of examples:

```
grep -c "/jsp/source.jsp?/jsp/ /jsp/source.jsp??"
access_log
grep "client denied" error_log | tail -n 10
```

The first example will list the number of attacks trying to exploit the Apache Tomcat Source.JSP Malformed Request Information Disclosure Vulnerability, the second example will list the ten last denied clients, for example:

```
[Thu Jul 11 17:18:39 2002] [error] [client
foo.example.com] client denied by server configuration:
/usr/local/apache/htdocs/.htpasswd
```

As you can see, the log files only report what already has happened, so if the client had been able to access the `.htpasswd` file you would have seen something similar to:

```
foo.example.com - - [12/Jul/2002:01:59:13 +0200] "GET
/.htpasswd HTTP/1.1"
```

in your Access Log. This means you probably commented out the following in your server configuration file:

```
<Files ".ht*">
    Require all denied
</Files>
```

Merging of configuration sections

The merging of configuration sections is complicated and sometimes directive specific. Always test your changes when creating dependencies on how directives are merged.

For modules that don't implement any merging logic, such as `mod_access_compat`, the behavior in later sections depends on whether the later section has any directives from the module. The configuration is inherited until a change is made, at which point the configuration is *replaced* and not merged.