

In this chapter, security means allowing people to see what you want them to see and preventing them from seeing what you don't want them to see. Additionally, there are the issues of what measures you need to take on your server in order to restrict access *via* non-Web means. This chapter illustrates the precautions you need to take to protect your server from malicious access and modification of your Web site.

The most common questions ask how to protect documents and restrict access. Unfortunately, because of the complexity of the subject and the nature of the Web architecture, these questions also tend to have the most complex answers or often no convenient answers at all.

Normal security nomenclature and methodology separate the process of applying access controls into two discrete steps; in the case of the Web, they may be thought of as the server asking itself these questions:

- Are you really who you claim to be?
- Are you allowed to be here?

These steps are called *authentication* and *authorization*, respectively. Here's a real-world example: a flight attendant checks your photo identification (authentication) and your ticket (authorization) before permitting you to board an airplane.

Authentication can be broken down into what might be called *weak* and *strong*. Weak authentication is based on the correctness of credentials that the end user supplies (which therefore may have been stolen from the real owner—hence the name “weak”), whereas strong authentication is based on attributes of the request over which the end user has little or no control, and it cannot change from request to request—such as the IP address of his system.

Although checking authentication and authorization are clearly separate activities, their application gets a bit blurred in the context of the Apache Web server modules. Even though the main difference between the many security modules is how they store the credentials (in a file, a database, an LDAP directory, etc.), they nevertheless have to provide the code to retrieve the credentials from the store, validate those supplied

by the client, and check to see if the authenticated user is authorized to access the resource. In other words, there's a lot of functionality duplicated from module to module, and although there are frequently similarities between their behavior and directives, the lack of shared code means that sometimes they're not quite as similar as you'd hope. This overloading of functionality has been somewhat addressed in the next version of the Web server after 2.0 (still in development at the time of this writing).

In addition to the matter of requiring a password to access certain content from the Web server, there is the larger issue of securing your server from attacks. As with any software, Apache has, at various times in its history, been susceptible to conditions that would allow an attacker to gain inappropriate control of the hosting server. For example, they may have been able to access, or modify, files that the site administrator had not intended to give access to, or they may have been able to execute commands on the target server. Thus, it is important that you know what measures need to be taken to ensure that your server is not susceptible to these attacks.

The most important measure that you can take is to keep apprised of new releases, and read the CHANGES file to determine if the new version fixes a security hole to which you may be subject. Running the latest version of the Apache server is usually a good measure in the fight against security vulnerabilities.

Recipes in this chapter show you how to implement some of the frequently requested password scenarios, as well as giving you the tools necessary to protect your server from external attacks.

Authentication and Authorization

When checking for access to restricted documents, there are actually two different operations involved: checking to see who you are and checking to see if you're allowed to see the document.

The first part, checking to see who you are, is called *authentication*. The Web server doesn't know who you are, so you need to provide some proof of your identity, such as a username and matching password. When the server successfully compares these bits of information (called *credentials*) with those in its databases, the server will proceed, but if you're not in the list, or the information doesn't match, the server will turn you away with an error status.

Once you have convinced the server you are who you say you are, it will look at the list of people allowed to access the document and see if you're on it; this is called *authorization*. If you are on the list, access proceed normally; otherwise, the server returns an error status and denies access.

The two different operations do not differentiate in the errors they return; it is always a 401 (unauthorized) code, even if the failure was in authentication. This is to prevent would-be attackers from being able to tell when they have valid credentials.

6.1 Using System Account Information for Web Authentication

Problem

You want all the users on your Unixish system to be able to authenticate themselves over the Web using their already-assigned usernames and passwords.

Solution

Set up a realm using *mod_auth* and name */etc/passwd* as the *AuthUserFile*:


```
<Directory "/home">
  AuthType Basic
  AuthName HomeDir
  AuthUserFile /etc/passwd
  Require valid-user
  Satisfy All
</Directory>
```

Discussion

We must stress that using system account information for Web authentication is a very bad idea, unless your site is also secured using SSL. For one thing, any intruder who happens to obtain one of your users' credentials not only can access the protected files over the Web, but can actually log onto your system where it's possible to do significant damage. For another, Web logins don't have the same security controls as most operating systems; over the Web, an intruder can keep hammering away at a username with password after password without the system taking any defensive measures; all *mod_auth* will do is record a message in the Apache error log. However, most operating systems will enter a paranoid mode and at least ignore login attempts for a while after some number of failures.

If you still want to do this, either because you consider the risk acceptable or because it doesn't apply in your situation, the *httpd.conf* directives in the Solution will do the trick. The syntax and order of the fields in a credential record used by *mod_auth* happens (and not by accident) to match the standard layout of the */etc/passwd* lines. *mod_auth* uses a simple text file format in which each line starts with a username and password and may optionally contain additional fields, with the fields delimited by colons. For example:

```
smith:$apr1$GLWeF/..$8h0XRFUpHhBJHp0UdNFe51
```

mod_auth ignores any additional fields after the password, which is what allows the */etc/passwd* file to be used. Note that the password in the example is encrypted. 

You can manage Apache *mod_auth* credential files with the *htpasswd* utility, but don't use this utility on the */etc/passwd* file! Use the normal account maintenance tools for that.

Note that this technique will not work if shadow passwords are in use, because the password field of */etc/passwd* contains nothing useful in that situation. Instead, the passwords are stored in the file */etc/shadow*, which is readable only by *root*, while Apache runs as an unprivileged user. Furthermore, most modern Unixish operating systems use the */etc/shadow* means of user authentication by default.

See Also

- Authentication and Authorization
- HTTP, Browsers, and Credentials
- Weak and Strong Authentication
- The *htpasswd* man page
- The *passwd(5)* man page

6.2 Setting Up Single-Use Passwords

Problem

You want to be able to provide credentials that will allow visitors into your site only once.

Solution

No solution is available with standard Apache features.

Discussion

As described in HTTP, Browsers, and Credentials, the concept of being “logged in” to a site is an illusion. In order to achieve the desired one-time-only effect, the server needs to complete the following steps:

1. Note the first time the user successfully presents valid credentials.
2. Somehow, associate that fact with the user’s “session.”
3. Never allow those credentials to succeed again if the session information is different from the first time they succeeded.

The last step is not a simple task, and it isn’t a capability provided in the standard Apache distribution. To complicate matters, there is the desire to start a timeout once the credentials have succeeded, so that the user doesn’t authenticate once and then leave his browser session open for days and retain access.

Fulfilling this need would require a custom solution. Unfortunately, we are not aware of any open or public modules that provide this capability; however, search and watch the module registry for possible third-party implementations.

See Also

- Recipe 6.3
- <http://modules.apache.org/>

6.3 Expiring Passwords

Problem

You want a user's username and password to expire at a particular time or after some specific interval.

Solution

No solution is available with standard Apache features, but a few third-party solutions exist.

Discussion

Refer to HTTP, Browsers, and Credentials. In order for Apache to provide this functionality, it would need to store more than just the valid username and password; it would also have to maintain information about the credentials' expiration time. No module provided as part of the standard Apache distribution does this.

There are several third-party solutions to this problem, including the Perl module *Apache::Htpasswd::Perishable* and the *mod_perl* handler *Apache::AuthExpire*.

There are two slightly different ways to look at this problem, which will influence your choice of a solution. You may want a user's authentication to be timed out after a certain amount of time, or perhaps after a certain period of inactivity, forcing them to log in again. Or you may want a particular username/password pair to be completely expired after a certain amount of time, so that it no longer works. The latter might be used instead of a single-use password, which is impractical to implement in HTTP.

Apache::Htpasswd::Perishable partially implements the latter interpretation of the problem by adding expiration information to the password file. Inheriting from the *Apache::Htpasswd* module, it adds two additional methods, *expire* and *extend*, which set an expiration date on the password and extend the expiration time, respectively.

For example, the following code will open a password file and set an expiration date on a particular user entry in that file:

```
use Apache::Htpasswd::Perishable;

my $pass = Apache::Htpasswd::Perishable->new("/usr/local/apache/passwords/user.pass")
    or die "Could not open password file.";
$pass->expire('waldo', 5); # Set the expiration date 5 days in the future
```

Such a mechanism is only useful if expired passwords are removed from the password file periodically. This can be accomplished by running the following *cron* script every day. This will delete those users for whom the expiration date has passed:

```
#!/usr/bin/perl
use Apache::Htpasswd::Perishable;

my $password_file = '/usr/local/apache/passwords/user.pass';

open(F,$password_file) or die "Could not open password file.";
my @users;
while (my $user = <F>) {
    $user =~ s/^([:])+:.*$/$1/;
    push @users, $user;
}
close F;

my $pass = Apache::Htpasswd::Perishable->new($password_file) or die
"Could not open password file.";
foreach my $user (@users) {
    $pass->htDelete($user) unless $pass->expire($user) > 0;
}
```

Apache::AuthExpire, by contrast, implements timeouts on “login sessions.” That is, a user must reauthenticate after a certain period of inactivity. This gives you protection against the user who steps away from her computer for a few moments, leaving herself “logged in.”

As previously discussed, HTTP is stateless and so does not really have a concept of being logged in. However, by watching repeated connections from the same address, such a state can be simulated.

To use the expiring functionality offered by *Apache::AuthExpire*, download the module from CPAN, and install it:

```
# perl -MCPAN -e shell
cpan> install Apache::AuthExpire
```

Then configure your Apache server to use this module for your authentication handler.

```
PerlAuthenHandler Apache::AuthExpire
PerlSetVar DefaultLimit 7200
```

The given example will time out idle connections after 7200 seconds, which is 2 hours.

See Also

- Recipe 6.2
- <http://modules.apache.org/>
- <http://search.cpan.org/author/JJHORNER/Apache-AuthExpire/AuthExpire.pm>
- <http://search.cpan.org/author/ALLENDAY/Apache-Htpasswd-Perishable/Perishable.pm>

6.4 Limiting Upload Size

Problem

With more and more Web hosting services allowing customers to upload documents, uploads may become too large. With a little creativity, you can put a limit on uploads by using the security capabilities of the server.

Solution

Assume you want to put a limit on uploads of ten thousand (10,000) bytes. The simplest way to do this is to add a *LimitRequestBody* directive to the appropriate scope:

```
<Directory "/usr/local/apache2/uploads">
    LimitRequestBody 10000
</Directory>
```

If a user tries to send a request that's too large, he'll get an error message. However, the default Apache message may leave something to be desired so you can either tailor it with a *ErrorDocument 413* directive, or with some more complex (and more flexible) jiggery-pokery such as the following:

```
SetEnvIf Content-Length "[1-9][0-9]{4,}" upload_too_large=1
<Location /upload>
    Order Deny,Allow
    Deny from env=upload_too_large
    ErrorDocument 403 /cgi-bin/remap-403-to-413
</Location>
```

You can tailor the response by making the */cgi-bin/remap-403-to-413* script look something like this:

```
#!/usr/local/bin/perl
#
# Perl script to turn a 403 error into a 413 IFF
# the forbidden status is because the upload was
# too large.
#
if ($ENV{'upload_too_large'}) {
    #
    # Constipation!
    #
    print <<EOHT
Status: 413 Request Entity Too Large
Content-type: text/plain; charset=iso-8859-1
Content-length: 84

    Sorry, but your upload file exceeds the limits
    set forth in our terms and conditions.
EOHT
}
else {
    #
```

```

# This is a legitimate "forbidden" error.
#
my $uri = $ENV{'REDIRECT_REQUEST_URI'};
my $length = 165 + length($uri);
print <<EOHT
Status: 403 Forbidden
Content-type: text/html; charset=iso-8859-1
Content-length: $length

<html>
<head>
  <title>Forbidden</title>
</head>
<body>
  <h1>Forbidden</h1>
  <p>
    You don't have permission to access $uri
    on this server.
  </p>
</body>
</html>
EOHT
}
exit(0);

```

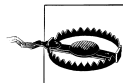
Discussion

monospace

This script is invoked when a request results in a **403 Forbidden** error (which is what the *Deny* directive causes if it's triggered). It checks to see if it's a real forbidden condition, or whether the upload file is too large, displaying an appropriate error page.

Note that both paths issue a **Status** CGI response header field; this is necessary to propagate the correct status back to the client. Without this, the status code would be 200 OK because the script would have been invoked successfully, which is hardly the appropriate status. An incorrect status code may cause the browser to report to the user that the file was uploaded successfully, which might generate confusion, as this may be in conflict with the message of the error page.

Actually there is a status value that corresponds to “you sent me something too large” (413), so we remap the *Deny*'s 403 (Forbidden) status to it.



The same **Content-length** field is used to indicate the amount of data included in a POST request, such as from a Web form submission, so be careful not to set your maximum too low or your forms may start getting this error!

See Also

- Chapter 9

6.5 Restricting Images from Being Used Off-Site

Problem

Other sites are linking to images on your system, stealing bandwidth from you and incidentally making it appear as though the images belong to them. You want to ensure that all access to your images is from documents that are on your server.

Solution

Add the following lines to the *.htaccess* file in the directory where the images are, or to the appropriate *<Directory>* container in the *httpd.conf* file. Replace the *myserver.com* with your domain name:

```
<FilesMatch "\.(j?pg|jpeg|gif|png)$">
    SetEnvIfNoCase Referer "^http://([^\/*]*\.)?myserver.com/" local_referrer=1
    Order Allow,Deny
    Allow from env=local_referrer
</FilesMatch>
```

In fact, by using the following recipe, you can even go one step further, and return a different image to users accessing your images *via* an off-site reference:

```
SetEnvIfNoCase Referer "^http://([^\/*]*\.)?myserver.com/" local_referrer=1
RewriteRule %{ENV:local_referrer} !=1 /Stolen-100x100.png [L]
```

RewriteCond "%{ENV:local_referrer}" !=1 "

Discussion

RewriteRule ".*" "/Stolen-100x100.png" [L]

monospace

The first solution will cause all requests for image files to be refused with a 403 Forbidden status unless the link leading to the request was in one of your own documents. This means that anyone linking to your images from a different Web site system will get the error instead of the image, because the referer does not match the approved server name.

Note that this technique can cause problems for requests that do not include a *Referer* request header field, such as people who visit your site through an anonymizing service or who have their browser configured not to send this information.

The second solution is similar to the first, except that it substitutes an image of your choice for the one requested, rather than denying the request. Using the values in the Solution, you can construct a *Stolen-100x100.png* that has whatever admonitory message or perhaps just some picture that will deter the visitor from “stealing” your images.



This technique has a tendency to get the problem fixed more quickly, since visitors to the thieving site will see “This Image Is Stolen!”—and that’s typically not the impression the site’s owners would like them to get. Simply returning a 403 (Forbidden) error will result in a broken-image icon on the referring page, and *everyone* is used to those nowadays, and thinks nothing of them.

See Also

- Recipe 6.21

6.6 Requiring Both Weak and Strong Authentication

Problem

You want to require both weak and strong authentication for a particular resource. For example, you wish to ensure that the user accesses the site from a particular location and to require that he provides a password.

Solution

Use the *Satisfy* directive to require both types of authentication:

```
<Directory /www/htdocs/sensitive>

    # Enforce all restrictions
    Satisfy All

    # Require a password
    AuthType Basic
    AuthName Sensitive
    AuthUserFile /www/passwords/users
    AuthGroupFile /www/passwords/groups
    Require group salesmen

    # Require access from a certain network
    Order deny,allow
    Deny from all
    Allow from 192.168.1
</Directory>
```

Discussion

In this example, a user must provide a login, identifying him as a member of the salesmen group, and he must also use a machine on the 192.168.1 network.

The *Satisfy All* directive requires that all access control measures be enforced for the specified scope. A user accessing the resource from a nonmatching IP address will immediately receive a *Forbidden* error message in his browser, while, in the logfile, the following error message is logged:

```
[Sun May 25 15:31:53 2003] [error] [client 208.32.53.7] client denied by server
configuration: /usr/local/apache/htdocs/index.html
```

Users who are in the required set of IP addresses, however, receive a password dialog box and are required to provide a valid username and password.

Looking forward a little bit, the syntax of this recipe will change somewhat with the 2.4 release of the web server. The *Allow*, *Deny*, and *Satisfy* directives have long confused Apache admins, and so this syntax has been revised.

The new syntax will look something like the following, with the old syntax still being available to those who want it by use of the *mod_access_compat* module.

The *Satisfy All* command will look like:

```
<SatisfyAll>
    Require group salesmen
    Require ip 192.168.1
</SatisfyAll>
```

See also the *SatisfyOne* directive, which replaces the *Satisfy any* syntax.

See Also

- Recipe 6.9

6.7 Managing .htpasswd Files

Problem

You wish to create password files for use with Basic HTTP authentication.

Solution

Use the *htpasswd* utility to create your password file, as in Table 6-1.

Table 6-1. Managing password files with *htpasswd*

Command	Action
% <u>htpasswd -c user.pass</u> waldo	Create a new password file called <i>user.pass</i> with this one new entry for user waldo. Will prompt for password.
% <u>htpasswd user.pass ralph</u>	Add an entry for user ralph in password file <i>user.pass</i> . Will prompt for password.
% <u>htpasswd -b user.pass</u> <u>ralph mydogspot</u>	Add a user ralph to password file <i>user.pass</i> with password mydogspot.

Or, use the Perl module *Apache::Htpasswd* to manage the file programmatically:

```
use Apache::Htpasswd;
$pass = new Apache::Htpasswd("/usr/local/apache/passwords/user.pass") or
    die "Couldn't open password file.";

# Add an entry
$pass->htpasswd("waldo", "emerson");
```

```
# Delete entry
$pass->htDelete("waldo");
```

Discussion

The *htpasswd* utility, which comes with Apache, is located in the *bin* subdirectory.



On some third-party distributions of Apache, the *htpasswd* program has been copied into a directory in your path, but ordinarily it will not be in your path; you will either have to put it there, or provide the full path to the program in order to run it, such as */usr/local/apache/bin/htpasswd*.

The first line of the Solution creates a new password file at the specified location. That is, in the example given, it creates a new password file called *user.pass*, containing a username and password for a user *waldo*. You will be prompted to enter the desired password, and then prompted to repeat the password for confirmation.

The **-c** flag creates a new password file, even if a file of that name already exists, so make sure that you only use this flag the first time. After that, using it causes your existing password file to be obliterated and replaced with the (almost empty) new one.

The second line in the Solution adds a password to an existing password file. As before, the user is prompted to enter the desired password, and then prompted to confirm it by typing it again.

The examples given here create a password file using the *crypt* algorithm by default on all platforms other than Windows, Netware, and TPF. On those platforms, the MD5 algorithm is used by default.

For platforms that use *crypt*, each line of the password file looks something like:

```
waldo:/z32oW/ruTI8U
```

The portion of the line following the username and colon is the encrypted password. Other usernames and passwords appear one per line.

The *htpasswd* utility provides other options, such as the ability to use the MD5 algorithm to encrypt the password (the **-m** flag), provide the password on the command line rather than being prompted for it (the **-b** flag), or print the results to *stdout*, rather than altering the password file (the **-n** flag). **monospace**

The **-b** flag can be particularly useful when using the *htpasswd* utility to create passwords in some scripted fashion, rather than from an interactive prompt. The third line of the recipe above illustrates this syntax.

As of Apache 2.0.46, the **-D** flag lets you delete an existing user from the password file:

```
% htpasswd -D user.pass waldo
```

whereas in previous versions, you would need to use some alternate method to remove lines from the file. For example, you could remove a line using *grep*, or simply open the file in a text editor:

```
% egrep -v '^waldo:' user.pass >! user.pass
```

Apache::Htpasswd, written by Kevin Meltzer, is available from CPAN (<http://cpan.org/>) and gives a Perl interface to Apache password files. This allows you to modify your password files from CGI programs or *via* other mechanisms, using just a few lines of Perl code as shown in the recipe.

In addition to the methods demonstrated in this recipe, there are also methods for checking a particular password against the contents of the password file, obtaining a list of users from the file, or retrieving the encrypted password for a particular user, among other things. See the documentation for this fine module for the full details on its many features.

One final note about your password file. We strongly recommend that you store your password file in some location that is not accessible through the Web (i.e., outside of your document directory). By putting it in your document directory, you run the risk of someone downloading the file and running a brute-force password cracking algorithm against it, which will eventually yield your passwords for them to use.

See Also

- Recipe 6.8
- <http://search.cpan.org/author/KMELTZ/Apache-Htpasswd/Htpasswd.pm>

6.8 Making Password Files for Digest Authentication

Problem

You need to create a password file to be used for Digest authentication.

Solution

Use the following command forms to set up a credential file for a realm to be protected by **Digest** authentication:

```
% htdigest -c "By invitation only" rbowen  
% htdigest "By invitation only" krietz
```

Discussion

Digest authorization, implemented by *mod_auth_digest*, uses an MD5 hash of the username, password, and authentication realm to check the credentials of the client. The *htdigest* utility, which comes with Apache, creates these files for you.

The syntax for the command is very similar to the syntax for the *htpasswd* utility, except that you must also specify the authentication realm that the password will be used for. The resulting file contains one line per user, looking something like the following:

```
rbowen:By invitation only:23bc21f78273f49650d4b8c2e26141a6
```

Note that, unlike entries in the password files created by *htpasswd*, which can be used anywhere, these passwords can be used only in the specified authentication realm, because the encrypted hash includes the realm.

As with *htpasswd*, the *-c* flag creates a new file, possibly overwriting an existing file. You will be prompted for the password and then asked to type it again to verify it.

htdigest does not have any of the additional options that *htpasswd* does.

See Also

- Recipe 6.7

6.9 Relaxing Security in a Subdirectory

Problem

There are times when you might want to apply a tight security blanket over portions of your site, such as with something like:

```
<Directory /usr/local/apache/htdocs/BoD>
    Satisfy All
    AuthUserFile /usr/local/apache/access/bod.htpasswd
    Require valid-user
</Directory>
```

Because of Apache's scoping rules, this blanket applies to all documents in that directory and in any subordinate subdirectories underneath it. But suppose that you want to make a subdirectory, such as *BoD/minutes*, available without restriction?

Solution

The *Satisfy* directive is the answer. Add the following to either the *.htaccess* file in the subdirectory or in an appropriate *<Directory>* container:

```
Satisfy Any
Order Deny,Allow
Allow from all
```

HTTP, Browsers, and Credentials

It is easy to draw incorrect conclusions about the behavior of the Web; when you have a page displayed in your browser, it is natural to think that you are still connected to that site. In actuality, however, that's not the case—once your browser fetches the page

from the server, both disconnect and forget about each other. If you follow a link, or ask for another page from the same server, a completely new exchange has begun.

When you think about it, this is fairly obvious. It would make no sense for your browser to stay connected to the server while you went off to lunch or home for the day.

Each transaction that is unique and unrelated to others is called *stateless*, and it has a bearing on how HTTP access control works.

When it comes to password-protected pages, the Web server doesn't remember whether you've accessed them before or not. Down at the HTTP level where the client (browser) and server talk to each other, the client has to prove who it is every time; it's the *client* that remembers your information.

When accessing a protected area for the first time in a session, here's what actually gets exchanged between the client and the server:

1. The client requests the page.
2. The server responds, "You are not authorized to access this resource (a 401 unauthorized status). This resource is part of authentication realm XYZ." (This information is conveyed using the WWW-Authenticate response header field; see RFC 2616 for more information.)
3. If the client isn't an interactive browser, at this point it probably goes to step 7. If it is interactive, it asks the user for a username and password, and shows the name of the *realm* the server mentioned.
4. Having gotten credentials from the user, the client reissues the request for the document—including the credentials this time.
5. The server examines the provided credentials. If they're valid, it grants access and returns the document. If they aren't, it responds as it did in step 2.
6. If the client receives the unauthorized response again, it displays some message about it and asks the user if he wants to try entering the username and password again. If the user says yes, the client goes back to step 3.
7. If the user chooses not to reenter the username and password, the client gives up and accepts the "unauthorized" response from the server.

Once the client has successfully authenticated with the server, it remembers the credentials, URL, and realm involved. Subsequent requests that it makes for the same document or one "beneath" it (e.g., */foo/bar/index.html* is "beneath" */foo/index.html*) causes it to send the same credentials automatically. This makes the process start at step 4, so even though the challenge/response exchange is still happening between the client and the server, it's hidden from the user. This is why it's easy to get caught up in the fallacy of users being "logged on" to a site.

This is how all HTTP weak authentication works. One of the common features of most interactive Web browsers is that the credentials are forgotten when the client is shut down. This is why you need to reauthenticate each time you access a protected document in a new browser session.

monospace

Discussion

This tells Apache that access is granted if the requirements of either the weak (user credentials) or strong protection (IP address) mechanisms are fulfilled. Then it goes on to say that the strong mechanism will always be happy regardless of the visitor's origin.

Be aware that this sets a new default security condition for all subdirectories below the one affected. In other words, you are not just unlocking the one subdirectory but all of its descendants as well.

See Also

- Recipe 6.6
- Recipe 6.10

6.10 Lifting Restrictions Selectively

Problem

You want most documents to be restricted, such as requiring a username and password, but want a few to be available to the public. For example, you may want *index.html* to be publicly accessible, whereas the rest of the files in the directory require password authentication.

Solution

Use the *Satisfy Any* directive in the appropriate place in your *.htaccess* or *httpd.conf* file:

```
<Files index.html>
    Order Deny,Allow
    Allow from all
    Satisfy Any
</Files>
```

You can locate this in a *.htaccess* file, or within a *<Directory>* container to limit its effect.

```
<Directory "/usr/local/apache/htdocs">
    Satisfy All
    Order allow,deny
    Deny from all
    <Files *.html>
        Order deny,allow
        Allow from all
        Satisfy Any
    </Files>
</Directory>
```


Discussion

Regardless of what sorts of restrictions you may have on other files, or on the directory as a whole, the `<Files>` container in the solution makes the `index.html` file accessible to everyone without limitation. *Satisfy Any* tells Apache that any of the restrictions in place may be satisfied, rather than having to enforce any particular one. In this case, the restriction in force will be *Allow from all*, which permits access for all clients.

This method can be easily extended to apply to arbitrary filename patterns using shell global characters. To extend it to use regular expressions for the filename, use the `<FilesMatch>` directive instead.

Weak and Strong Authentication

The basic Apache security model for HTTP is based on the concepts of *weak* and *strong* authentication mechanisms. Weak mechanisms are those that rely on information volunteered by the user; strong ones use credentials obtained without asking him. For instance, a username and password constitute a set of weak credentials, whereas the IP address of the user's client system is regarded as a strong one.

One difference between the two types lies in how Apache handles an authentication failure. If invalid weak credentials are presented, the server will respond with a 401 Unauthorized status, which allows the user to try again. In contrast, a failure to authenticate when strong credentials are required will result in a 403 Forbidden status—for which there is no opportunity to retry.

In addition, strong and weak credentials can be required in combination; this is controlled by the *Satisfy* directive. The five possible requirements are:

- None. No authentication required.
- Only strong credentials are needed.
- Only weak credentials are required.
- Both strong and weak credentials are accepted; if either is valid, access is permitted.
- Both strong and weak credentials are required.

See Also

- Recipe 6.9
- Recipe 6.6
- http://httpd.apache.org/docs/mod/mod_access.html

6.11 Authorizing Using File Ownership

Problem

You wish to require user authentication based on system file ownership. That is, you want to require that the user that owns the file matches the username that authenticated.

Solution

Use the *Require file-owner* directive:

```
<Directory /home/*/public_html/private>
    AuthType Basic
    AuthName "MyOwnFiles"
    AuthUserFile /some/master/authdb
    Require file-owner
</Directory>
```

Discussion

The goal here is to require that username `jones` must authenticate in order to access the `/home/jones/public_html/private` directory.

The user does not authenticate against the system password file but against the *AuthUserFile* specified in the example. Apache just requires that the name used for authentication matches the name of the owner of the file or directory in question. Note also that this is a feature of *mod_auth* and is not available in other authentication modules.



This feature was added in Apache 1.3.22. In Apache 2.2, this functionality is provided by the module *mod_authz_owner*.

See Also

- The *Require file-group* keyword at http://httpd.apache.org/docs/mod/mod_auth.html#require

6.12 Storing User Credentials in a MySQL Database

Problem

You wish to use user and password information in your MySQL database for authenticating users.

Solution

For Apache 1.3, use *mod_auth_mysql*:

```
Auth_MySQL_Info db_host.example.com db_user my_password
Auth_MySQL_General_DB auth_database_name

<Directory /www/htdocs/private>
    AuthName "Protected directory"
    AuthType Basic
    require valid-user
</Directory>
```

For Apache 2.2 and later, use *mod_authn_dbi*:

```
AuthnDbiDriver Config1 mysql
AuthnDbiHost Config1 db.example.com
AuthnDbiUsername Config1 db_username
AuthnDbiPassword Config1 db_password
AuthnDbiName Config1 auth_database_name
AuthnDbiTable Config1 auth_database_table
AuthnDbiUsernameField Config1 user_field
AuthnDbiPasswordField Config1 password_field
AuthnDbiIsActiveField Config1 is_active_field

AuthnDbiConnMin Config1 3
AuthnDbiConnSoftMax Config1 12
AuthnDbiConnHardMax Config1 20
AuthnDbiConnTTL Config1 600

<Directory "/www/htdocs/private">
    AuthType Digest
    AuthName "Protected directory"
    AuthBasicProvider dbi
    AuthnDbiServerConfig Config1
    Require valid-user
</Directory>
```

Discussion

There are a number of modules called *mod_auth_mysql*. The module used in the previous example is the *mod_auth_mysql* from http://www.diegonet.com/support/mod_auth_mysql.shtml. For the full explanation of the database fields that you will need to create, and the additional options that the module affords, you should consult the documentation on the Web site.

If you are running Apache 2.2 or later, you will want to take advantage of the new authentication framework, and use the module *mod_authn_dbi*, available from http://open.cyanworlds.com/mod_authn_dbi/. Because of the new authentication API in Apache 2.2, a number of things are possible that were not possible in earlier versions. For example, a single module, such as *mod_authn_dbi*, can be used for either Basic or Digest authentication, by simply changing the *AuthType* directive from *Basic* to *Di-*

gest. (*AuthBasicProvider* would also become *AuthDigestProvider* in the previous example.)

mod_authn_db uses *libdbi*, which is a generic database access library, allowing you to use your favorite database server to provide authentication services. *libdbi* drivers are available for most popular database servers. For a more complete description of *mod_authn_db*, you should consult the documentation on the Web site.

See Also

- http://www.diegonet.com/support/mod_auth_mysql.shtml
- http://open.cyanworlds.com/mod_authn_db/

6.13 Accessing the Authenticated Username

Problem

You want to know the name of the user who has authenticated.

Solution

Some scripting modules, such as *mod_php*, provide a standard interface for accessing values set by the server. For instance, to obtain the username that was used to authenticate from within a PHP script, it would access a field in the `$_SERVER` superglobal array:

```
$auth_user = $_SERVER['REMOTE_USER'];
```

For a Perl or *mod_perl* script, use

```
my $username = $ENV{REMOTE_USER};
```

In a Server-Side Include (SSI) directive, this may look like:

```
Hello, user <!--#echo var="REMOTE_USER" -->. Thanks for visiting.
```

Other scripting modules may provide specific means of accessing this information. For those that don't, or for nonscript applications, use the appropriate means to consult the `REMOTE_USER` environment variable.

Discussion

When a user has authenticated, the environment variable `REMOTE_USER` is set to the name with which she authenticated. You can access this variable in CGI programs, SSI directives, PHP files, and a variety of other methods. The value also will appear in your *access_log* file.

Note that although it is the convention for an authentication module to set this variable, there are reportedly some third-party authentication modules that do not set it but provide other methods for accessing that information.

See Also

- Recipe 6.14

6.14 Obtaining the Password Used to Authenticate

Problem

You want to get the password the user used to authenticate.

Solution

Standard Apache modules do not make this value available. It is, however, available from the Apache API if you wish to write your own authentication methods.

In the Apache 1.3 API, you need to investigate the `ap_get_basic_auth_pw` function. In the 2.0 API, look at the `get_basic_auth` function.

If you write an authentication handler with *mod_perl*, you can retrieve the username and password with the `get_username` function:

```
my ($username, $password) = get_username($r);
```

You can make this information available to CGI scripts executed by the server if you rebuild the package with the appropriate flag. For Apache 1.3 and 2.0,

```
% CFLAGS="$CFLAGS -DSECURITY_HOLE_PASS_AUTHORIZATION"
```

Discussion

For security reasons, although the username is available as an environment variable, the password used to authenticate is *not* available in any simple manner. The rationale behind this is that it would be a simple matter for unscrupulous individuals to capture passwords so that they could then use them for their own purposes. Thus, the decision was made to make passwords near to impossible to obtain.

The only way to change this is to rebuild the server from the sources with a particular (strongly discouraged) compilation flag. Alternately, if you write your own authentication module, you would of course have access to this value, as you would need to verify it in your code.

See Also

- Recipe 6.13

6.15 Preventing Brute-Force Password Attacks

Problem

You want to disable a username when there are repeated failed attempts to authenticate using it, as if it is being attacked by a password-cracker.

Solution

There is no way to do this with standard Apache authentication modules. The usual approach is to watch your logfile carefully. Or you can use something like *Apache::BruteWatch* to tell you when a user is being attacked:

```
PerlLogHandler Apache::BruteWatch
PerlSetVar BruteDatabase DBI:mysql:brute1og
PerlSetVar BruteDataUser username
PerlSetVar BruteDataPassword password

PerlSetVar BruteMaxTries 5
PerlSetVar BruteMaxTime 120
PerlSetVar BruteNotify rbowen@example.com
```

Discussion

Because of the stateless nature of HTTP and the fact that users are not, technically, “logged in” at all (see HTTP, Browsers, and Credentials), there is no connection between one authentication attempt and another. Most Apache auditing tools, such as *mod_security*, work on a per-request basis, and have no way to compare one request to another to build a profile across multiple requests. This makes it possible to repeatedly attempt to log in with a particular username, without it being easily detectable by any automated tool.

Your best bet is to carefully watch the log files, or to have some process which watches the log files for you.

Apache::BruteWatch is one way to watch the logfile and send notification when a particular account is being targeted for a brute-force password attack. With the configuration shown above, if a given account fails authentication five times in 2 minutes, the server administrator will be notified of the situation, so that she can take appropriate measures, such as blocking the offending address from the site.

Apache::BruteWatch is available from CPAN (CPAN.org) and requires *mod_perl* to run.

We are not, at this time, aware of another module that does this in real time.

See Also

- HTTP, Browsers, and Credentials

6.16 Using Digest Versus Basic Authentication

Problem

You want to understand the distinction between the **Basic** and **Digest** authentication methods.

Solution

Use *AuthType Basic* and the *htpasswd* tool to control access using **Basic** authentication. Use *AuthType Digest* and the *htdigest* tool for the **Digest** method.

Discussion

Basic Web authentication is exactly that: primitive and insecure. It works by encoding the user credentials with a reversible algorithm (essentially base-64 encoding) and transmitting the result in plaintext as part of the request header. Anyone (or anything) that intercepts the transmission can easily crack the encoding of the credentials and use them later. As a consequence, **Basic** authentication should only be used in environments where the protected documents aren't truly sensitive or when there is no alternative.

In contrast, **Digest** authentication uses a more secure method that is much less susceptible to credential theft, spoofing, and replay attacks. The exact details don't matter; the essential ingredient is that no username or password traverses the network in plaintext.

Preparing a realm to use **Basic** authentication consists of simply storing the username/password pair and telling the server where to find them. The password may or may not be encrypted. The same credentials may be applied to any realm on the server, or even copied to a completely different server and used there. They may be stored in a variety of databases; multiple modules exist for storing **Basic** credentials in flat text files, GDBM files, MySQL databases, LDAP directories, and so on.

Setting up **Digest** authentication is a little more involved. For one thing, the credentials are not transportable to other realms; when you generate them, you specify the realm to which they apply. For another, the only storage mechanism currently supported directly by the Apache package is flat text files; if you want to keep your **Digest** credentials in an LDAP directory or Oracle database, you're going to have to look for third-party modules to do it or else write one yourself.

In addition to the more complex setup process, **Digest** authentication currently suffers from a lack of market penetration. That is, even though Apache supports it, not all browsers and other Web clients do; so you may end up having to use **Basic** authentication simply, because there's nothing else available to your users. However, this is less and less the case with the passage of time, and there are very few Web clients in the wild any more that don't support **Digest** authentication.

See Also

- Recipe 6.18

6.17 Accessing Credentials Embedded in URLs

Problem

You know people access your site using URLs with embedded credentials, such as *http://user:password@host/*, and you want to extract them from the URL for validation or other purposes.

Solution

None; this is a nonissue that is often misunderstood.

Discussion

Embedding the username and password in the URL gives a way to distribute a link to your users to access a password-protected site directly, without being prompted for the password. However, what tends to be misunderstood about this is that the username and password are actually sent to the server in the ordinary way (i.e., *via* the `WWW-Authenticate` header) and not as part of the URL. The browser dissects the URL and turns it into the appropriate request header fields to send to the server.

6.18 Securing WebDAV

Problem

You want to allow your users to upload and otherwise manage their Web documents with WebDAV but without exposing your server to any additional security risks.

Solution

Require authentication to use WebDAV:

```
<Directory "/www/htdocs/dav-test">
  Order Allow,Deny
  Deny from all
  AuthDigestFile "/www/acl/.htpasswd-dav-test"
  AuthDigestDomain "/dav-test/"
  AuthName "DAV access"
  Require valid-user
  Satisfy Any
</Directory>
```


Discussion

Because WebDAV operations can modify your server's resources and *mod_dav* runs as part of the server, locations that are WebDAV-enabled need to be writable by the user specified in the server's *User* directive. This means that the same location is writable by any CGI scripts or other modules that run as part of the Apache server. To keep remote modification operations under control, you should enable access controls for WebDAV-enabled locations. If you use weak controls, such as user-level authentication, you should use **Digest** authentication rather than **Basic**, as shown in the Solution.

The contents of the `<Directory>` container could be put into a *dav-test/.htaccess* file, as well. Note that the authentication database (specified with the *AuthDigestFile* directive) is not within the server's URI space, and so it cannot be fetched with a browser nor with any WebDAV tools.

Your authentication database and *.htaccess* files should not be modifiable by the server user; you don't want them getting changed by your WebDAV users!

See Also

- Recipe 6.16

6.19 Enabling WebDAV Without Making Files Writable by the Web User

Problem

You want to run WebDAV but don't want to make your document files writable by the Apache server user.

Solution

Run two web servers as different users. The DAV-enabled server, for example, might run as *User dav*, *Group dav*, whereas the other server, which is responsible for serving your content, might run as *User nobody*, *Group nobody*. Make the Web content writable by the *dav* user, or the *dav* group.



Remember that only a single Web server can be handling a particular port/IP address combination. This means that your WebDAV-enabled server will have to be using either a different address, a different port, or both than the non-WebDAV server.

Discussion

A big security concern with DAV is that the content must be modifiable by the Web server user for DAV to be able to update that content. This means that any content also can be edited by CGI programs, SSI directives, or other programs running under the Web server. Although the Apache security guidelines caution against having any files writable by the web server user, DAV requires it.

By running two Apache servers, you can move around this limitation. The DAV-enabled web server, running on an alternate port, has the *User* and *Group* directives set to an alternate user and group, such as:

```
User dav
Group dav
```

which is the owner of the Web content in question. The other Web server, which will be responsible for serving content to users, runs as a user who does not have permission to write to any of the documents.

The DAV-enabled Web server should be well authenticated, so that only those who are permitted to edit the site can access that portion of the server. You should probably also set up this server to be very lightweight, both in the modules that you install as well as in the number of child processes (or threads) that you run.

Finally, it should be noted that the *perchild* MPM, under Apache 2.0, supports the idea of running different virtual hosts with different user ids, so that this recipe could be accomplished by enabling DAV just for the one particular vhost. However, as of this writing, the *perchild* MPM is not working yet.

See Also

- http://httpd.apache.org/docs-2.0/mod/mod_dav.html
- <http://httpd.apache.org/docs-2.0/mod/perchild.html>

6.20 Restricting Proxy Access to Certain URLs

Problem

You don't want people using your proxy server to access particular URLs or patterns of URLs (such as MP3 or streaming video files).

Solution

You can block by keyword:

```
ProxyBlock .rm .ra .mp3
```

You can block by specific backend URLs:

```
<Directory proxy:http://other-host.org/path>
  Order Allow,Deny
  Deny from all
  Satisfy All
</Directory>
```

Or you can block according to regular expression pattern matching:

```
<Directory proxy:*>
  RewriteEngine On
  #
  # Disable proxy access to Real movie and audio files
  #
  RewriteRule "\.(rm|ra)$" "-" [F,NC]
  #
  # Don't allow anyone to access .mil sites through us
  #
  RewriteRule "[a-z]+://[-.a-z0-9]*\.mil($|/)" "-" [F,NC]
</Directory>
```

Discussion

All of these solutions will result in a client that attempts to access a blocked URL receiving a 403 Forbidden status from the server.

The first solution uses a feature built into the proxy module itself: the *ProxyBlock* directive. It's simple and efficient, and it catches the results so that future accesses to the same URL are blocked with less effort; however, the pattern matching it can perform is extremely limited and prone to confusion. For instance, if you specify:

```
ProxyBlock .mil
```

the server denies access to both *http://www.navy.mil/* and *http://example.com/spec.mil/list.html*. This is probably not what was intended!

The second method allows you to impose limitations based on the URL being fetched (or gateway, in the case of a *ProxyPass* directive).

The third method, which allows more complex what-to-block patterns to be constructed, is both more flexible and more powerful, and somewhat less efficient. Use it only when the other methods prove insufficient.



<DirectoryMatch> containers work as well, so more complex patterns may be used.

The flags to the *RewriteRule* directive tell it, first, that any URL matching the pattern should result in the server returning a 403 Forbidden error (F for forbidden), and second that the pattern match is case-insensitive (NC or nocase).

monospace

One disadvantage of the *mod_rewrite* solution is that it can be too specific. The first *RewriteRule* pattern can be defeated if the client specifies path-info or a query string, or if the origin server uses a different suffix naming scheme for these types of files. A little cleverness on your part can cover these sorts of conditions, but beware of trying to squeeze too many possibilities into a single regular expression pattern. It's generally better to have multiple *RewriteRule* directives than to have a single all-singing all-dancing one that no one can read—and is, hence, prone to error.

See Also

- The *mod_proxy* and *mod_rewrite* documentation at http://httpd.apache.org/docs/mod/mod_proxy.html and http://httpd.apache.org/docs/mod/mod_rewrite.html



6.21 Protecting Files with a Wrapper

Problem

You have files to which you want to limit access using some method other than standard Web authentication (such as a members-only area).

Solution

In *httpd.conf*, add the following lines to a *<Directory>* container whose contents should be accessed only through a script:

```
RewriteEngine On
RewriteRule "\.(dll|zip|exe)$" protect.php [NC]
RewriteCond %{REMOTE_ADDR} "!^my.servers.ip"
RewriteRule "\.cgi$" protect.php [NC]
```

And an example *protect.php* that just displays the local URI of the document that was requested:

```
<?php
/*
 * The URL of the document actually requested is in
 * $_SERVER['REQUEST_URI']. Appropriate decisions
 * can be made about what to do from that.
 */
Header('Content-type: text/plain');
$body = sprintf("Document requested was: %s\n", $_SERVER['REQUEST_URI']);
Header('Content-length: ' . strlen($body));
print $body;
?>
```

Discussion

In the situation that prompted this recipe, authentication and authorization were completed using a cookie rather than the standard mechanisms built into the Web proto-

cols. Any request for a document on the site was checked for the cookie and redirected to the login page if it wasn't found, was expired, or had some other problem causing its validity to be questioned.

This is fairly common and straightforward. What is needed in addition to this is a way to limit access to files according to the cookie and ensure that no URL-only request could reach them.

To this end, a wrapper is created (called *protect.php* in the Solution), which is invoked any time one of the protected document types is requested. After validating the cookie, the *protect.php* script figures out the name of the file from the environment variables, determines the content-type from the extension, and opens the file and sends the contents.

This is illustrated in the Solution. Any time a document ending in one of the extensions *.dll*, *.zip*, *.exe*, or *.cgi* is requested from the scope covered by the *mod_rewrite* directives, and the request comes from some system other than the Web server system itself (i.e., from a client system), the *protect.php* script will be invoked instead. In the Solution, the script simply displays the local URI of the document that is requested; applying additional access control or other functionality is easily developed from the example.

If access control is the main purpose of the wrapper and the access is granted, the wrapper needs to send the requested document to the client. In this case, the wrapper could either determine the filesystem path to the desired document and use the PHP routine `fpass thru()` to open it and send it to the client, or it could access the document using PHP's ability to open a URL as though it were a file with the `fopen("http://docurl")` function call. (This latter method is necessary if the document requires server processing, such as if it's a script.)

This would ordinarily trigger the wrapper on the dynamic document again, causing a loop. To prevent this, the wrapper is only applied to dynamic documents if the requesting host isn't the server itself. If it is the Web server making the request, we know the wrapper has already been run and you don't need to run it again. The server processes the document as usual and sends the contents back to the wrapper, which is still handling the original request, and it dutifully passes it along to the client. This is handled by the *RewriteCond* directive, which says "push requests for scripts through the wrapper unless they're coming from the server itself."

This method is perhaps a little less than perfectly elegant and not the best for performance, because each CGI request involves at least two concurrent requests, but it *does* address the problem.

See Also

- Chapter 5

6.22 Protecting Server Files from Malicious Scripts

Problem

Scripts running on your Web server may access, modify, or destroy files located on your Web server if they are not adequately protected. You want to ensure that this cannot happen.

Solution

Ensure that none of your files are writable by the `nobody` user or the `nobody` group, and that sensitive files are not readable by that user and group:

```
# find / -user nobody  
# find / -group nobody
```

Discussion

The *User* and *Group* directives specify a user and group under whose privileges the Web server will run. These are often set to the values of `nobody` and `nobody`, respectively, but they can vary in different setups. It is often advisable to create a completely new user and group for this purpose, so that there is no chance that the user has been given additional privileges of which you are not aware.

Because everything runs with these privileges, any files or directories that are accessible by this user and/or group will be accessible from any script running on the server. This means that a script running under one virtual host may possibly modify or delete files contained within another virtual host, either intentionally or accidentally, if those files have permissions making this possible.

Ideally, no files anywhere on your server should be owned by, or writable by, the server user, unless for the explicit purpose of being used as a datafile by a script. And, even for this purpose, it is recommended that a real database be used, so that the file itself cannot be modified by the server user. And if files simply must be writable by the server, they should definitely not be in some Web-accessible location, such as `/cgi-bin/`.

See Also

- Recipe 8.13
- Recipe 6.23

6.23 Setting Correct File Permissions

Problem

You want to set file permissions to provide the maximum level of security.

Solution

The *bin* directory under the *ServerRoot* should be owned by user root, group root, and have file permissions of 755 (**rwxr-xr-x**). Files contained therein should also be owned by root.root and be mode 755.

Document directories, such as *htdocs*, *cgi-bin*, and *icons*, will have to have permissions set in a way that makes the most sense for the development model of your particular Web site, but under no circumstances should any of these directories or files contained in them be writable by the Web server user.



The solution provided here is specific to Unixish systems. Users of other operating systems should adhere to the principles laid out here, although the actual implementation will vary.

The *conf* directory should be readable and writable only by root, as should all the files contained therein.

The *include* and *libexec* directories should be readable by everyone, writable by no one.

The *logs* directory should be owned and writable by root. You may, if you like, permit other users to read files in this directory, as it is often useful for users to be able to access their logfiles, particularly for troubleshooting purposes.

The *man* directory should be readable by all users.

Finally, the *proxy* directory should be owned by and writable by the server user.



On most Unixish file systems, a *directory* must have the x bit set in order for the files therein to be visible.

Discussion

You should be aware that if you ask 12 people for the correct ways to set file permissions on your Apache server, you will get a dozen different answers. The recommendations here are intended to be as paranoid as possible. You should feel free to relax these recommendations, based on your particular view of the world and how much you trust your users. However, if you set file permissions any more restrictive than this, your Apache server is likely not to function. There are, of course, exceptions to this, and cases in which you could possibly be more paranoid are pointed out later.

The most important consideration when setting file permissions is the Apache server user—the user as which Apache runs. This is configured with the *User* and *Group* directives in your *httpd.conf* file, setting what user and group the Apache processes will

run as. This user needs to have read access to nearly everything but should not have write access to anything.

The recommended permissions for the *bin* directory permit anyone to run programs contained therein. This is necessary in order for users to create password files using the *htpasswd* and *htdigest* utilities, run CGI programs using the *suexec* utility, check the version of Apache using *httpd -v*, or use any of the other programs in this directory. There is no known security risk of permitting this access. The Web server itself cannot be stopped or started by an unprivileged user under normal conditions. These files, or the directory, should never be writable by nonroot users, as this would allow compromised files to be executed with root privileges.

Extra-paranoid server administrators may wish to make the *bin* directory, and its contents, readable and executable only by root. However, the only real benefit to doing so is that other users cannot run the utilities or *httpd* server, such as on a different port. Some of those utilities, such as *htpasswd* and *htdigest*, are intended to be run by content providers (i.e., users) in addition to the Webmaster.

The *conf* directory, containing the server configuration files, can be locked down as tightly as you like. Although it is unlikely that reading the server configuration files will allow a user to gain additional privileges on the server, more information is always useful for someone trying to compromise your server. You may, therefore, wish to make this directory readable only by root. However, most people will consider this just a little too paranoid.

Document directories are particularly problematic when it comes to making permission recommendations, as the recommended setting will vary from one server to another. On a server with only one content provider, these directories should be owned by that user and readable by the Apache user. On a server with more than one content developer, the files should be owned by a group of users who can modify the files but still be readable by the Apache user. The *icons* directory is a possible exception to this rule, because the contents of that directory are rarely modified and do not need to be writable by any users.

The *include* and *libexec* directories contain files that are needed by the Apache executable at runtime and only need to be readable by root, which starts as root, and by no other users. However, since the *include* directory contains C header files, it may occasionally be useful for users to have access to those files to build applications that need those files.

The *logs* directory should under no circumstances ever be writable by anyone other than root. If the directory is ever writable by another user, it is possible to gain control of the Apache process at start time and gain root privileges on the server. Whether you permit other users to read files in this directory is up to you and is not required. However, on most servers, it is very useful for users to be able to access the logfiles—particularly the *error_log* file, in order to troubleshoot problems without having to contact the server administrator.

The *man* directory contains the manpages for the various utilities that come with Apache. These need to be readable by all users. However, it is recommended that you move them to the system *man* path, or install them there when you install Apache by providing an argument to the `--mandir` argument specifying the location of your system *man* directory.

Finally, the *proxy* directory should be owned by, and writable by, the server user. This is the only exception to the cardinal rule that nothing should be writable by this user. The *proxy* directory contains files created by and managed by *mod_proxy*, and they need to be writable by the unprivileged Apache processes. If you are not running a proxy server with *mod_proxy*, you may remove this directory entirely.

See Also

- *Learning the Unix Operating System*, Fifth Edition, by Jerry Peek, Grace Todino, and John Strang (O'Reilly)
- http://www.onlamp.com/pub/a/bsd/2000/09/06/FreeBSD_Basics.html

6.24 Running a Minimal Module Set

Problem

You want to eliminate all modules that you don't need in order to reduce the potential exposure to security holes. What modules do you really need?

Solution

For Apache 1.3, you can run a bare-bones server with just three modules. (Actually, you can get away with not running any modules at all, but it is not recommended.)

```
% ./configure --disable-module=all --enable-module=dir \  
> --enable-module=mime --enable-module=log_config \  

```

For Apache 2.x, this is slightly more complicated, as you must individually disable modules you don't want:

```
% ./configure --disable-access \  
> --disable-auth --disable-charset-lite \  
> --disable-include --disable-log-config --disable-env --disable-setenvif \  
> --disable-mime --disable-status --disable-autoindex --disable-asis \  
> --disable-cgid --disable-cgi --disable-negotiation --disable-dir \  
> --disable-imap --disable-actions --disable-alias --disable-userdir
```

Note that with 2.x, as with 1.3, you may wish to enable *mod_dir*, *mod_mime*, and *mod_log_config*, by simply leaving them off of this listing.

Discussion

A frequent security recommendation is that you eliminate everything that you don't need; if you don't need something and don't use it, then you are likely to overlook security announcements about it or forget to configure it securely. The question that is less frequently answered is exactly what you do and don't need.

A number of Apache package distributions come with everything enabled, and people end up running modules that they don't really need—or perhaps are not even aware that they are running.

This recipe is an attempt to get to the very smallest Apache server possible, reducing it to the minimum set of modules that Apache will run. That is, if you take any of these out, Apache will not even start up, let alone serve a functional Web site.

Apache 1.3

With Apache 1.3, this question is fairly easy to answer. We've reduced it to a set of three modules, and, actually, you can eliminate all of the modules if you really want to, as long as you're aware of the implications of doing so.

mod_dir is the module that takes a request for / and turns it into a request for */index.html*, or whatever other file you have indicated with the *DirectoryIndex* directive as the default document for a directory. Without this module, users typing just your hostname into their browser will immediately get a 404 error, rather than a default document. Granted, you could require that users specify a hostname and filename in their URL, in which case you could dispense with this module requirement. This would, however, make your Web site fairly hard to use.

mod_mime enables Apache to determine what MIME type a particular file is, and send the appropriate MIME header with that file, enabling the browser to know how to render that file. Without *mod_mime*, your Web server will treat *all* files as having the MIME type set by the *DefaultType* directive. If this happens to match the actual type of the file, well and good; otherwise, this will cause the browser to render the document incorrectly. If your Web site consists only of one type of files, you can omit this module.

Finally, *mod_log_config*, while not technically required at all, is highly recommended. Running your Web server without any activity logfiles will leave you without any idea of how your site is being used, which can be detrimental to the health of your server. However, you should note that it is not possible to disable the *ErrorLog* functionality of Apache, and so, if you really don't care about the access information of your Web site, you could feasibly leave off *mod_log_config* and still have error log information.



The default distributed configuration file will need some adjustment to run under these reduced conditions. In particular, you will probably need to remove *Order*, *Allow*, and *Deny* directives (provided by *mod_access*), and you will need to remove *LogFormat* and *CustomLog* directives if you remove *mod_log_config*. Many other sections of the configuration files are protected by *<IfModule>* sections and will still function in the absence of the required modules.

Apache 2.x

With Apache 2.x, a new configuration utility is used, and so the command-line syntax is more complicated. In particular, there is no single command-line option to let you remove all modules, and so every module must be specified with a `-disable` directive.

The list of modules that are minimally required for Apache 2.x is the same as that for 1.3. *mod_dir*, *mod_mime*, and *mod_log_config* are each recommended, but not mandated, for the same reasons outlined previously.

6.25 Restricting Access to Files Outside Your Web Root

Problem

You want to make sure that files outside of your Web directory are not accessible.

Solution

For Unixish systems:

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

For Windows systems:

```
<Directory C:/>
  Order deny,allow
  Deny from all
  AllowOverride None
  Options None
</Directory>
```

Repeat for each drive letter on the system.

Discussion

Good security technique is to deny access to everything, and then selectively permit access where it is needed. By placing a *Deny from all* directive on the entire filesystem,

you ensure that files cannot be loaded from any part of your filesystem unless you explicitly permit it, using a *Allow from all* directive applied to some other `<Directory>` section in your configuration.

If you wanted to create an *Alias* to some other section of your filesystem, you would need to explicitly permit this with the following:

```
Alias /example /var/example
<Directory /var/example>
    Order allow,deny
    Allow from all
</Directory>
```

See Also

- http://httpd.apache.org/docs/mod/mod_access.html

6.26 Limiting Methods by User

Problem

You want to allow some users to use certain methods but prevent their use by others. For instance, you might want users in group A to be able to use both GET and POST but allow everyone else to use only GET.

Solution

Apply user authentication *per* method using the *Limit* directive:

```
AuthName "Restricted Access"
AuthType Basic
AuthUserFile filename
Order Deny,Allow
Allow from all
<Limit GET>
    Satisfy Any
</Limit>
<LimitExcept GET>
    Satisfy All
    Require valid-user
</Limit>
```

Discussion

It is often desirable to give general access to one or more HTTP methods, while restricting others. For example, although you may wish any user to be able to GET certain documents, you may wish for only site administrators to POST data back to those documents.

It is important to use the *LimitExcept* directive, rather than attempting to enumerate all possible methods, as you're likely to miss one.

See Also

- http://httpd.apache.org/docs/mod/mod_auth.html
- http://httpd.apache.org/docs/mod/mod_access.html
- <http://httpd.apache.org/docs/mod/core.html#limit>
- <http://httpd.apache.org/docs/mod/core.html#limitexcept>

6.27 Restricting Range Requests

Problem

You want to prevent clients from requesting partial downloads of documents within a particular scope, forcing them to request the entire document instead.

Solution

You can overload *ErrorDocument* [403](#) to make it handle range requests. To do this, put the following into the appropriate *<Directory>* container in your *httpd.conf* file or in the directory's *.htaccess* file:

```
SetEnvIf "Range" "." partial_requests
Order Allow,Deny
Allow from all
Deny from env=partial_requests
ErrorDocument 403 /forbidden.cgi
```

Then put the following into a file named *forbidden.cgi* in your server's *DocumentRoot*:

```
#!/usr/bin/perl -w
use strict;
my $message;
my $status_line;
my $body;
my $uri = $ENV{'REDIRECT_REQUEST_URI'} || $ENV{'REQUEST_URI'};
my $range = $ENV{'REDIRECT_HTTP_RANGE'} || $ENV{'HTTP_RANGE'};
if (defined($range)) {
    $body = "You don't have permission to access "
        . $ENV{'REQUEST_URI'}
        . " on this server.\r\n";
    $status_line = '403 Forbidden';
}
else {
    $body = "Range requests disallowed for document '"
        . $ENV{'REQUEST_URI'}
        . "'.\r\n";
    $status_line = '416 Range request not permitted';
}
print "Status: $status_line\r\n"
    . "Content-type: text/plain;charset=iso-8859-1\r\n"
    . "Content-length: " . length($body) . "\r\n"
    . "\r\n"
```

```
. $body;  
exit(0);
```

Or use *mod_rewrite* to catch requests with a Range header. To do this, put the following into the appropriate *<Directory>* container in your *httpd.conf* file or in the directory's *.htaccess* file:

```
RewriteEngine On  
RewriteCond "%{HTTP:Range}" ""  
RewriteRule "(.*)" "/range-disallowed.cgi" [L,PT]
```

Then put the following into a file named *range-disallowed.cgi* in your server's *DocumentRoot*:

```
#!/usr/bin/perl -w  
use strict;  
my $message = "Range requests disallowed for document '"  
    . $ENV{'REQUEST_URI'}  
    . "'\r\n";  
print "Status: 416 Range request not permitted\r\n"  
    . "Content-type: text/plain; charset=iso-8859-1\r\n"  
    . "Content-length: " . length($message) . "\r\n"  
    . "\r\n"  
    . $message;  
exit(0);
```

Discussion

Both of these solutions are a bit sneaky about how they accomplish the goal.

The first overloads an *ErrorDocument 403* script so that it handles both real “access forbidden” conditions *and* range requests. The *SetEnvIf* directive sets the *partial_request* environment variable if the request header includes a Range field, the *Deny* directive causes the request to be answered with a 403 Forbidden status if the environment variable is set, and the *ErrorDocument* directive declares the script to handle the 403 status. The script checks to see whether there was a Range field in the request header so it knows how to answer—with a “you can’t do Range requests here” or with a real “document access forbidden” response.

The second solution uses *mod_rewrite* to rewrite any requests in the scope that include a Range header field to a custom script that handles only this sort of action; it returns the appropriate status code and message. The “sneaky” aspect of this solution is rewriting a valid and successful request to something that forces the response status to be *unsuccessful*.

See Also

- http://httpd.apache.org/docs/mod/mod_setenvif.html
- http://httpd.apache.org/docs/mod/mod_access.html
- http://httpd.apache.org/docs/mod/mod_rewrite.html

6.28 Rebutting DoS Attacks with *mod_evasive*

Problem

You want to protect your server from Denial of Service (DoS) attacks.

Solution

Obtain *mod_evasive* from http://www.zdziarski.com/projects/mod_evasive/ and use a configuration like the following

```
DOSPageCount      2
DOSPageInterval   1
DOSSiteCount      50
DOSSiteInterval   1
DOSBlockingPeriod 10
```

Discussion

mod_evasive is a third-party module that performs one simple task, and performs it very well. It detects when your site is receiving a Denial of Service (DoS) attack, and it prevents that attack from doing as much damage as it would do if left to run its course.

“Denial of Service” is a fairly broad term used to refer to attacks that consist of connections that cause your server to be so busy handling them that it can’t do anything else, such as talk to legitimate clients. Usually, in the case of Apache, this consists of simply making hundreds of HTTP requests per second. Often, the attacker will not even wait for the response, but will immediately disconnect and make another request, leaving Apache making a response to a client that isn’t even there any more.

mod_evasive detects when a single client is making multiple requests in a short period of time, and denies further requests from that client. The period for which the ban is in place can be very short, because it just gets renewed the next time a request is detected from that same host.

This configuration places two restrictions on requests. First, the *DOSPage* directives state that if a single client address requests the same URL more than twice in a single second, it should be blocked. The *DOSSite* directives state that if a single client address requests more than 50 URLs in a single second, it should be blocked. This second value is higher because sometimes a single page will contain a large number of images, and so will result in a larger number of requests from one client.

The *DOSBlockingPeriod* directive sets the interval for which the client will be blocked—in this case, 10 seconds. Although this seems like a very short interval, it can stretch indefinitely, as each time that same client attempts to connect (and is blocked) the denial period will start over again.

6.29 chrooting Apache with *mod_security*

Problem

You want to chroot Apache to make it more secure.

Solution

There are a number of different ways to chroot Apache. One of the simplest is to use *mod_security*, and add the following directive:

```
SecChrootDir /chroot/apache
```

Discussion

chroot is a Unix command which causes a program to run in a jail. That is to say, when the command is launched, the accessible file system is replaced with another path, and the running application is forbidden to access any files outside of its new file system. By doing this, you are able to control what resources the program has access to, and prevent it from writing to files outside of that directory, or running any programs that are not in that directory. This prevents a large number of exploits by simply denying the attacker access to the necessary tools.

The trouble with *chroot* is that it is very inconvenient. For example, when you chroot Apache, you must copy into the new file system any and all libraries or other files that Apache needs to run. For example, if you're running *mod_ssl*, you'd need to copy all of the OpenSSL libraries into the chroot jail so that Apache could access them. And if you had Perl CGI programs, you'd need to copy Perl, and all its modules, into the chroot directory.

mod_security gets around this complexity by chrooting Apache, not when it starts up, but immediately before it forks its child processes. This solves the *mod_ssl* problem mentioned above, but it would not solve the Perl problem, because the Perl CGI program is run by the forked child process. However, the number of things that you'll need to move or copy into the chroot jail is greatly reduced, and tends to consist only of things that you're running as CGI programs, rather than all of the libraries that Apache needs while it is starting up. This greatly reduced complexity increases the probability that someone would actually chroot Apache, as otherwise the complexity is such that most of us would never be willing to put up with the inconvenience.

If you're running Apache 1.3, you'll need to make sure that *mod_security* appears first in your *LoadModule* list, so that it can have the necessary level of control over how things are orchestrated. It's important that the chrooting happen at just the right moment, and in order for this to happen, *mod_security* needs to get there first before another module can take over.

If you're running Apache 2.0, this isn't necessary, as the module "knows" when it's supposed to load, and the right thing happens.

See Also

- <http://modsecurity.org/>

6.30 Migrating to 2.2 Authentication

Problem

You had authentication working, and then you moved to Apache 2.2, and everything is different.

Solution

Authentication was rearchitected in Apache 2.2 to more completely separate authentication and authorization as separate steps which can be configured independently. Although, at first, it can seem to be change for the sake of change, once you understand this separation, the new configuration syntax makes a lot more sense, and the changes that you'll have to make seem more sensible.

Discussion

These terms—authentication and authorization—are defined in some detail in the introduction to this chapter. Traditionally, Apache has blurred the boundary between these two concepts, making it difficult to configure one without being compelled to configure the other a particular way. For example, if you wanted to use Digest authentication, you were required to use a plain text file for the list of users. This is no longer the case with Apache 2.2.

monospace

In order to configure authentication and authorization in Apache 2.2, you'll need to make three decisions, which, in practical terms, involves choosing one module from each of three lists.

First, you'll need to determine which authentication type you're going to use. Your choices are Digest and Basic authentication, so you'll need to choose either *mod_auth_basic* or *mod_auth_digest*. This is done with the *AuthType* directive, as it was in versions before 2.2.

`AuthType Basic`

Next, you'll need to choose your authentication provider. This means that you're choosing where your authentication information will be stored—in a text file, dbm file, database, and so on. The directive for making this choice will be either *AuthBasicProvider*, if you're using Basic authentication, or *AuthDigestProvider*, if you're using Digest authentication.

monospace

monospace

```
AuthBasicProvider dbm
```

Finally, you'll need to provide some authorization requirement, using either a group or a some other method, using the *Require* directive.

```
Require user sarah
```

Thus, as compared to before 2.2, your authentication/authorization configuration directives may be an extra line or two:

```
AuthName "Private"  
AuthType Basic  
AuthBasicProvider dbm  
AuthDBMUserFile /www/passwords/passwd.dbm  
Require user sarah isaiah
```

See Also

- <http://httpd.apache.org/docs/2.2/howto/auth.html>

6.31 Blocking Worms with *mod_security*

Problem

You want to use the *mod_security* third-party module to intercept common probes before they actually reach your Web server's pages.

Solution

If you have *mod_security* installed (see Recipe 2.9), then you can use its basic "core rules" accessory package to intercept many of the most common attack and probe forms that hit web servers. The core rules package is periodically updated to keep pace with new issues that appear on the Web.

Discussion

Installing the core rules package and following the instructions in the *README* file makes this very simple. In addition, the files in the package make it easy to write your own rules by illustrating the formats.

See Also

- <http://modsecurity.org/projects/rules/>

6.32 Mixing Read-Only and Write Access to a Subversion Repository

Problem

You want to protect different portions of your Subversion repository differently, allowing read-access in some paths and write-access in others.

Solution

For a simple solution, you can use the `<LimitExcept>` to protect certain files or paths such that write access requires authentication:

```
<Location "/repos">
  DAV svn
  SVNParentPath "/repository/subversion"
  AuthType Basic
  AuthName "Log in for write access"
  AuthUserFile "/path/to/authfile"
  <LimitExcept GET REPORT OPTIONS PROPFIND>
    Require valid-user
  </LimitExcept>
</Location>
```

The configuration fragment above applies the restriction to the entire Subversion repository. For more flexible or fine-grained control, combine this with the `mod_authz_svn` module:

```
LoadModule authz_svn_module modules/mod_authz_svn.so
```

```
<Location "/repos">
  DAV svn
  SVNParentPath "/repository/subversion"
  Order Deny,Allow
  Allow from all
  AuthName "Log in for write access"
  AuthType Digest
  AuthDigestDomain "/repos/"
  AuthDigestFile "/path/to/digest-file"
  AuthzSVNAccessFile "/path/to/access-file"
  <Limit GET PROPFIND OPTIONS REPORT>
    Satisfy Any
  </Limit>
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Satisfy All
    Require valid-user
  </LimitExcept>
</Location>
```

Not sure what that colon there is supposed to mean. Please remove it.

Discussion

The first solution takes a simple approach: it says, in essence, “These methods are harmless, but if you use any others you gotta log in.”

The second solution combines this with the functionality of the *mod_authz_svn* module, which allows you to grant (or deny) access selectively according to the path involved. It still provides read access to the entire repository; if you want to limit that to only specific paths according to the username being used, remove the `<LimitExcept>` and `</LimitExcept>` lines, and remove the `<Limit>` container entirely. Then users will be required to log in to access the repository at all, and what they can access—read, write, or not at all—is defined by the SVN auth file identified by the *AuthzSVNAccessFile* directive.

See Also

- <http://httpd.apache.org/docs/2.0/core.html#limitexcept>
- <http://svnbook.red-bean.com/en/1.0/ch06s04.html>

6.33 Using Permanent Redirects to Obscure Forbidden URLs

Problem

When access to a file is forbidden, you don’t want the user’s browser to show its URL.

Solution

Add an *ErrorDocument* script that issues a permanent redirect to a “document not found” message page.

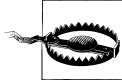
```
Alias "/not-found" "/path/to/documentroot/not-found.html"
ErrorDocument 403 "/cgi/handle-403"
```

and in the *cgi-bin/handle-403* script, something like this:

```
#!/usr/bin/perl -w
#
# Force a permanent redirect
#
print "Location: http://example.com/not-found\r\n\r\n";
exit(0);
```

Discussion

Ordinarily, when access to a document is forbidden, the browser’s display of its URL remains when the error is displayed. By using the steps in the Solution, the URL of the actual document being forbidden will be obscured by the server handling it with a redirection—which causes the browser to change its location bar—rather than as a normal error.



Note that the title of this recipe uses the verb “obscure.” That’s for a good reason; what’s being practiced here is called “security by obscurity,” which means “they can still get at it if they know exactly what to look for, but we’ll hope they don’t know it.” In a way, it’s like sticking your head in the sand and hoping that either the problem will go away, or that no one will discover it. A savvy user may be able to examine the network traffic and find out the name of the file being forbidden to him.

Because the redirection to the *not-found.html* file is successful, the browser is unaware that it tried to do anything wrong. You can spice this up a bit by making the redirection target a script that uses

```
print "Status: 403 Forbidden\r\n\r\n";
```

What will happen in this case is that the browser will request a forbidden file, the server will answer “go look over there,” and when the browser obediently looks “over there,” *then* it gets the 403 **Forbidden** error. But the browser’s location field will show the *not-found.html* document’s URL rather than that of the actual forbidden document.

See Also

- <http://httpd.apache.org/docs/2.0/core.html#errordocument>
- RFC 3875 sections 6.3.2 and 6.3.3 (<ftp://ftp.rfc-editor.org/in-notes/rfc3875.txt>)