CHAPTER 13

The PerlAccessHandler, PerlAuthenHandler, and PerlAuthzHandler

Introduction

Sooner or later, you're going to have to deal with the arduous task of maintaining various levels of resource security for your site. Apache and mod_perl are ready for you, providing hooks into the three distinct levels of resource control present in the Apache request lifecycle—the access control, authentication, and authorization phases of the request.

Adding your own security-related code to these hooks allows you to write very complex applications without polluting your content handlers with redirects, user name checks, and other cruft. In addition, the power and flexibility provided by mod_perl means you can limit access to server resources on an incredibly granular level. Your choice of access criteria is limited only by your imagination and coding skill.

The first phase of the trio is the `PerlAccessHandler`. This is generally used to control resource access based on server settings or other easily gathered information, such as the `User-Agent` request header, incoming IP address, or perhaps the existence of a specific file on the server. `PerlAccessHandlers` are not restricted in where they can appear in your configuration, and all configured handlers are run until one returns something other than `OK` or `DECLINED`. The typical return value for denying access to a resource is `FORBIDDEN`.

Next comes the `PerlAuthenHandler`, whose sole job is to determine the identity of a user. This only means that the credentials submitted by a client must match those held on the server (or other authenticating servers, as you will see in recipes later in this chapter). Specifically, it says nothing about whether the user has the proper privileges to access the requested resource, only that the server may proceed with additional checks based on properties of the authenticated user.

Like the `PerlAccessHandler`, the `PerlAuthenHandler` can appear anywhere within a configuration. The big difference is that no configured `PerlAuthenHandlers` will run unless the container directive governing the request is configured with the `Require` core Apache directive. Although the `AuthName` and `AuthType` directives are not required, Apache may fail to authenticate if they are not present, so typically mod_perl programmers insert placeholders for these values even if they are not going to use them within their `PerlAuthenHandler`.

Additionally, `PerlAuthenHandlers` have a winner-take-all attitude, so the first handler to return `OK` is sufficient to halt processing for the phase. Essentially, returning `OK` means that the user has been authenticated successfully, so no other authentication handlers need be concerned. Returning `AUTH_REQUIRED` means that a user has failed authentication.

After the user is authenticated, Apache gives you the option of restricting resource access based on this newfound knowledge. You might use this option to limit availability for administration portions of a site to a certain pool of users, for example. The `PerlAuthzHandler` follows the same rules as the `PerlAuthenHandler` with respect to its return values and its need for the `Require` directive and friends.

Together, these three handlers provide an extremely powerful and flexible way to pick and choose who can have access to your resources. The following recipes show how to leverage this power, from simple authentication all the way to adding your own authentication method to Apache.

13.1. Simple Access Control

You want to dynamically control access to server resources based on IP address or some other criterion.

Technique

Define and use a `PerlAccessHandler` to control access to the server resource.

```
package Cookbook::WormsBeGone;

use Apache::Constants qw(OK FORBIDDEN DECLINED SERVER_ERROR);

use strict;

sub handler {

  my $r = shift;

  my $ip  = $r->connection->remote_ip;
  my $uri = $r->uri;

  my $bad_ip_dir = $r->dir_config->get('BadIPdir');
  my @bad_urls   = $r->dir_config->get('BadURLs');

  # Do not run if no directory defined.
  return DECLINED unless ($bad_ip_dir);
  return FORBIDDEN if (-f "$bad_ip_dir/$ip");

  foreach my $bad_url (@bad_urls) {
    if (index($uri, $bad_url) == 0) {
      # Request is from a worm or Script Kiddie.
      # Create a file for the IP address.
      open(TOUCHFILE, ">$bad_ip_dir/$ip") or return SERVER_ERROR;
      close(TOUCHFILE);
      return FORBIDDEN;
    }
  }

  return OK;
}
1;
```

Comments

You can satisfy most of your simple access control needs by using a `PerlAccessHandler`, which is the first of the resource control phases. A `PerlAccessHandler` generally returns either `OK` or `FORBIDDEN`, depending on the circumstances. Apache iterates through all configured PerlAccessHandlers (as well as any C modules configured for the access control phase) until one returns an error code such as FORBIDDEN or SERVER_ERROR, at which point Apache takes the appropriate action. Returning OK simply runs the next PerlAccessHandler or, if no more handlers are configured, allows the request to proceed to the next phase.

The access checking code in our example is rather simple, but most PerlAccessHandlers are because they have only a limited amount of information available upon which to make decisions. For instance, mod_access supplies the Allow and Deny directives, which provide a simple access filter based on IP address. Our similarly simple example, Cookbook::WormsBeGone, uses a PerlAccessHandler to control access based on requests for common Internet worm URLs. Requests for blacklisted URLs result in a 403 Forbidden response. Furthermore, we remember the offending IP address that made the request and restrict all subsequent requests from that IP address. To use this handler, edit your httpd.conf file and add the following directives:

```
PerlSetVar BadIPdir /usr/local/apache/conf/bad_ips
PerlSetVar BadURLs /default.ida
PerlAddVar BadURLs /cgi-bin/phf

PerlAccessHandler Cookbook::WormsBeGone
```

To use this module, we first define a BadIPdir. This directory will contain blacklisted IP addresses represented as filenames. This directory needs to be writable by the user that owns the child httpd process (typically nobody). Next, we define one or more forbidden URLs by setting BadURLs. In this example, we define /default.ida, which corresponds to the *Code Red* worm, and /cgi-bin/phf, which corresponds to an Apache security hole dating back to 1996. By using a PerlAddVar we can easily add more worm URL targets as they appear. Next we configure the PerlAccessHandler outside of any container directive so it will run for every request.

The code first checks for a file in the BadIPdir directory corresponding to the client's IP address and immediately returns FORBIDDEN if present. Next, the request is checked against any defined BadURLs. If it matches, we blacklist the IP address, create a file in the BadIPdir corresponding to the IP address, and return FORBIDDEN. The end result is that we protect our site from infected systems, and can even manually forbid a remote IP by merely creating a file in the BadIPdir.

This simple example illustrates the basic concepts of a PerlAccessHandler, but we could do so much more with this power. The following list of customizations is easy to accomplish using mod_perl, and should give you an idea of the power available to you:

- Check IP addresses against `tcp_wrappers` or other databases of verboten addresses.
- Use a real back-end database to store the bad IP addresses.
- Enable checks for malformed `Host`, `User-Agent`, or `Cookie` headers.
- Page or e-mail an administrator whenever we receive an attack.

13.2. Restricting Access to Greedy Clients

Your Web site performance is suffering whenever a robot or other "greedy" Web client sends many requests in a short period of time.

Technique

Apply a throttling agent such as `Stonehenge::Throttle` or mod_throttle.

Comments

Like being popular in high school, being popular on the Internet is a good thing. In most cases, a high number of requests directly translates into an increase in business, money-generating transactions, or visibility—all of which are desirable if the requests are legitimate, and an absolute horror if the requests are coming from a blood-sucking robot whose only role in your business seems to be reducing the performance of your application.

In cases like these, installing some kind of throttling agent capable of detecting and taming those "users" who are eating up your bandwidth faster than you can make a sucking sound pays. One nice solution is `Stonehenge::Throttle`, a module by Randal Schwartz that has appeared on the mod_perl mailing list and in *Linux Magazine*. You can see the code at `http://www.stonehenge.com/merlyn/LinuxMag/col17.html`. `Stonehenge::Throttle` is basically a `PerlAccessHandler` that limits clients based on IP and the amount of CPU that they draw over a measured period of time. Although the code is quite lengthy, it uses a number of interesting techniques that ought to be easy to decipher if you have followed the recipes in earlier parts of this book. The reports from people who have implemented it have been very favorable and, best of all, it's written in Perl, so hacking it to suit your needs is easy.

If your site gets some really heavy traffic, you might want to consider using a C-based throttle to improve the overall performance of your server. mod_throttle, an Apache C extension module, is another throttling agent that comes packed with interesting and useful features. It allows you to define throttling policies based on the number of concurrent requests for a particular resource, bytes transferred over a time period, or the delay between requests. It also includes its own statistics content handler and fancy red, yellow, and green status indicators.

A few other Apache C modules also exist that can help you get a handle on where your bandwidth is going, such as mod_throttle_access and mod_bandwidth. One thing is clear—if you have a Web site with any amount of visibility, you will want to consider some type of throttling mechanism, lest you fall victim to the *Slashdot* effect, and your site is slowed to a crawl before you have your morning jolt of coffee.

13.3. Basic Authentication

You want to use the Basic authentication mechanism provided by Apache but want to store the user data in something other than a flat file.

Technique

Use the various mod_perl methods that hook into the Apache authentication API to write a `PerlAuthenHandler`.

```
package My::Authenticate;

use Apache::Constants qw(OK DECLINED AUTH_REQUIRED);
use My::Utils qw(authenticate_user);

use strict;

sub handler {

  my $r = shift;

  # Let subrequests pass.
  return DECLINED unless $r->is_initial_req;

  # Get the client-supplied credentials.
  my ($status, $password) = $r->get_basic_auth_pw;
  return $status unless $status == OK;

  # Perform some custom user/password validation.
  return OK if authenticate_user($r->user, $password);

  # Whoops, bad credentials.
  $r->note_basic_auth_failure;
  return AUTH_REQUIRED;
}
1;
```

This `PerlAuthenHandler` could then be configured using directives similar to

```
PerlModule My::Authenticate

<Location /private>
  PerlAuthenHandler My::Authenticate

  AuthType Basic
  AuthName "My Private Documents"
  Require valid-user
</Location>
```

Comments

When Apache receives a request governed by a `Require` directive, it enables the authentication phase. Typically, this phase involves a dialogue between the client and mod_auth that triggers the little browser pop-up box asking for a username and password. Although you might not have realized it until now, by the time you see the pop-up box, a request has already come full circle—the browser initiated a request that Apache denied, and the pop-up box is the browser's interpretation of the denial. After you enter a username and password, the browser will pass this information along with every subsequent request to the same authentication realm. The initial authorization process, then, actually involves (at least) two separate requests, each of which can be broken down into a bit more detail

The first request is typically intercepted by mod_auth who, on seeing no end-user credentials to check, returns a `401 Authorization Required` status back to the browser along with a `WWW-Authenticate` response header. This header marks the start of the challenge/response authorization cycle between the client and server.

The `WWW-Authenticate` header initiates the cycle with a server challenge that contains two bits of information: the authentication scheme the server expects to use, and the name of the authentication realm. These correspond to the values of the configured `AuthType` and `AuthName` directives governing the requested resource.

The presence of the `WWW-Authenticate` response header triggers the browser to prompt the user for a set of credentials. Armed with this new information, the browser issues its original request augmented with a response to the server challenge: an `Authorization` request header containing the user credentials. A typical challenge/response dialogue might look like the following:

```
GET /private/index.html HTTP/1.1
Accept: text/html, image/png, image/jpeg, image/gif, image/x-xbitmap, */*
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Accept-Language: en
Connection: Keep-Alive
Host: www.example.com
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 2000) Opera 5.12

HTTP/1.1 401 Authorization Required
WWW-Authenticate: Basic realm="My Private Documents"
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

GET /private/index.html HTTP/1.1
Accept: text/html, image/png, image/jpeg, image/gif, image/x-xbitmap, */*
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Accept-Language: en
Authorization: Basic YXV0aG9yOmxvb2tpbmcgZm9yIGFuIGVhc3RlciBlZ2csIGVoPw==
Connection: Keep-Alive, TE
Host: www.example.com
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 2000) Opera 5.12

HTTP/1.1 200 OK
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/plain
```

Although the headers are slightly different when proxies are involved, the mechanics of the challenge and response are the same.

The Authorization header contains the user-supplied credentials wrapped up in a base64-encoded string, which is not secure by any means but serves the purpose of transmitting the data. Typically, the encoded credentials are then parsed by mod_auth and compared against an encrypted password residing on the server. The AuthUserFile directive specifies the location of the file containing the username and encrypted password and is created by the htpasswd utility. It contains a series of entries similar to

```
author:KQ69wwvCZSWew
```

After the credentials have been verified against the AuthUserFile, mod_auth returns OK, and notes that the user successfully authenticated and no other authentication handler needs to run. Otherwise, 401 Authorization Required and the same WWW-Authenticate header are passed back to the client and the cycle continues.

Although a flat file is convenient for small numbers of users, adding a large number of users to the AuthUserFile is both inefficient and difficult to maintain. For more scalable solutions, standard Apache modules exist, such as mod_auth_db and mod_auth_dbm. Both enable you to store user credentials using Berkeley DB or DBM database files. These other modules are able to easily hook into the rather complex realm of authentication header parsing and generation by using an Apache API that makes the challenge/response cycle trivial to navigate and program. Of course, mod_perl passes the Apache API on to you, so yet another option is to write your own PerlAuthenHandler, as we did in our example, and insert the authorization mechanism of your choosing.

The Apache API significantly simplifies the challenge/response cycle; you need to understand only two methods when implementing a custom PerlAuthenHandler. The first is get_basic_auth_pw(), which does the job of parsing the Authorization request header. It returns both a status and the plain-text version of the supplied password. If the status is anything other than OK, your handler ought to propagate that value back to Apache; most of the time this is AUTH_REQUIRED, which initiates the challenge/response process for the client.

get_basic_auth_pw() also sets the value of $r->user() to the username gleaned from the base64-encoded authentication string. With this and the end user–supplied password in hand, you can implement any credential verification scheme you want. If the user passes muster, then simply return OK, which terminates the authentication phase of the request and allows Apache to proceed to the next phase. If the credentials are not valid, then the proper course of action is to call note_basic_auth_failure(), which sets the WWW-Authenticate response header to the appropriate value, and return

AUTH_REQUIRED. The cycle will then continue until the user is authenticated or calls it quits.

There is one twist to this theme that you will see in many PerlAuthenHandlers. Depending on your application, you might or might not want to check the value returned by $r->is_initial_req() to let subrequests pass by unauthenticated. This is entirely a matter of choice and is largely dependent on how your application uses subrequests.

The flexibility of using the mod_perl API to interact with the Basic authentication mechanism ought to be easy to see. You can authenticate based on the standard htpasswd generated file or use any one of a number of other data sources, such as a database, /etc/passwd, Radius or LDAP servers, and so on. In fact, a plethora of PerlAuthenHandlers is actively maintained on CPAN that implement just about any authentication variant you can dream up. Be sure to check there before you try to write your own handler; likely, some open-source module exists that you can use instead.

13.4. Setting User Credentials

You want to change or set the login name or password for a Basic authentication request.

Technique

Alter the incoming Authorization header before calling get_basic_auth_pw() or starting a subrequest.

```
package Cookbook::DefaultLogin;

use Apache::Constants qw(OK DECLINED);

use MIME::Base64 ();
use Socket qw(sockaddr_in inet_ntoa);

use strict;

sub handler {

  my $r = shift;
  my $c = $r->connection;

  # Parse the header ourselves.
  my $auth_header = $r->headers_in->get('Authorization');
  my $credentials = (split / /, $auth_header)[-1];
  my ($user, $passwd) = split /:/, MIME::Base64::decode($credentials), 2;

  # Make sure usernames are lowercase.
  $user = lc($user);

  # Automatic login for the user guest, or
  # localhost (makes telnet debugging easy).
  my $local_ip = inet_ntoa((sockaddr_in($c->local_addr))[1]);

  if ($user eq 'guest' || $c->remote_ip eq $local_ip) {
    $user   = $r->dir_config->get('DefaultUser');
    $passwd = $r->dir_config->get('DefaultPassword');
  }

  return DECLINED unless $user;  # nothing to do...

  # Re-join user and password and set the incoming header.
  $credentials = MIME::Base64::encode(join(':', $user, $passwd));

  $r->headers_in->set(Authorization => "Basic $credentials");

  return OK;
}
1;
```

Comments

As is the case for much of mod_perl, you can tinker with the Apache request cycle to accomplish almost anything. By adding a few lines of code, you can alter the login name or password the user entered, or even automatically use an alternate user for certain conditions.

The example handler `Cookbook::DefaultLogin` does two interesting things. First, it ensures that the username is always lowercase. Next, it checks for the special login name `guest` or if the connection is coming from `localhost`. If either is true, we permit access by modifying the username. By doing this, we can give out a guest account whose privileges mirror that of a real user without giving away a real username and password to third parties. The guest privileges can be changed or removed as quickly as we can alter our `httpd.conf`. Allowing `localhost` to pass unauthenticated is terribly convenient when a `telnet` debugging session is required for parts of a site under resource control.

To enable this behavior, simply add the `PerlAccessHandler` to your `httpd.conf`. The following example configuration shows `Cookbook::DefaultLogin` being applied to your run-of-the-mill Apache Basic authentication block:

```
<Location /main_bridge>
  AuthType Basic
  AuthName CaptainsOnly
  AuthUserFile /usr/local/apache/conf/htpasswd

  Require valid-user

  PerlAccessHandler Cookbook::DefaultLogin

  PerlSetVar  DefaultUser ishmael
  PerlSetVar  DefaultPassword whale
</Location>
```

With this configuration, when a user logs in as `guest` or the request comes from the `localhost`, we automatically authenticate as the user `ishmael` with the password `whale`.

The module works by modifying the headers of the current request. The first step involves pulling out the `Authorization` header. If it's present, we pull the credentials out and find the supplied username and password. The first modification we make is lowercasing the username, which is easily done. Next we check whether the username is `guest` or whether the connection is coming from the same IP as the server. If either is true, we create and encode a new `Authorization` header that contains the modified data, set it on the current request, and let Apache deal with the aftermath.

13.5. Conditional Authentication

You want to make authentication dynamic and conditional, only performing it under specific circumstances.

Technique

Configure your Apache as you would for normal authentication, but set the `PerlAuthenHandler` and `PerlAuthzHandler` handler stacks to `OK` when you do not want authentication to run.

First, configure your `httpd.conf`

```
<Location /YachtClub>
  SetHandler perl-script
  PerlHandler My::HoleInTheWater

  PerlSetVar PassIP 10.3.2.
  PerlAddVar PassIP 10.3.4.
  PerlAccessHandler Cookbook::PassLocalIP

  PerlAuthenHandler Apache::AuthenLDAP
  PerlAuthzHandler  Apache::AuthzLDAP

  AuthType Basic
  AuthName sloop
  Require blue-blazer
</Location>
```

Then, create a package to control the authentication phases.

```
package Cookbook::PassLocalIP;

use Apache::Constants qw(OK);

use strict;

sub handler {

  my $r = shift;

  # We don't need to do anything if Apache is going to
  # skip authentication anyway.
  return OK unless $r->some_auth_required;

  # Get the list of IP masks to allow.
  my @IPlist = $r->dir_config->get('PassIP');
  if (grep {$r->connection->remote_ip =~ m/^\Q$_/} @IPlist) {
    # Disable authentication if coming from an allowed IP...
    $r->set_handlers(PerlAuthenHandler => [\&OK]);

    # ... and disable authorization if that's also configured
    $r->set_handlers(PerlAuthzHandler =>[\&OK])
      if grep { lc($_->{requirement}) ne 'valid-user' } @{$r->requires};
  }

  return OK;
}
1;
```

Comments

As mentioned in the Introduction, the `Require` directive is necessary in order for Apache to run the authentication and authorization phases. Unfortunately, this situation makes it impossible to completely configure these phases on-the-fly using `push_handlers()` or `set_handlers()`—if there is no `Require` directive, then you can push a handler onto the `PerlAuthenHandler` stack all day, and it will not make a bit of difference.

The solution code offers one technique for conditionally configuring `PerlAuthenHandlers` and/or `PerlAuthzHandlers`. This technique might be used to remedy a situation where you want open access to a set of resources from within a local intranet but want to restrict access for users coming in from beyond the firewall. The controlling handler is installed as a `PerlAccessHandler`. As mentioned in Recipe 13.1, `PerlAccessHandlers` are for making decisions about access based on properties of the client or server—properties such as IP addresses, local file permissions, and the like—so it is arguably the logical place to control conditional authentication.

The first thing to do is see whether the authentication phases will run at all for the current URI. `some_auth_required()` will return true if any value is set for the `Require` directive, which lets us know whether Apache intends to actually run the authentication and authorization phases; if there is nothing to trigger these phases there is nothing for us to circumvent. When it has been determined that Apache will attempt to authenticate the user, we can apply a criterion that determines whether the user needs to go through the authentication process.

After it has been decided that the request is allowed to proceed without being challenged, we can skip over the authentication phase by using `set_handlers()` to abandon the current PerlAuthenHandler configuration and set the entire phase to simply return OK. Note that the value supplied to set_handlers() is the code reference for the constant subroutine Apache::Constants::OK. Because the first authentication handler to return OK wins, and mod_perl handlers typically run prior to any configured C modules like mod_auth, the authentication phase is essentially terminated before it begins.

Things work similarly for the authorization phase, which only needs to be disabled if the Require directive is set to anything other than valid-user. This is checked using the values returned from the requires() method. See the next recipe for a more detailed explanation of requires() and the authorization process.

13.6. User Authorization

You want to restrict server resources at the per-user level.

Technique

Write a PerlAuthzHandler for fine control over your server resources.

```
package Cookbook::AuthzRole;

use Apache::Constants qw(OK AUTH_REQUIRED);

use DBI;

use strict;

sub handler {

  my $r = shift;

  my $dbuser = $r->dir_config('DBUSER');
  my $dbpass = $r->dir_config('DBPASS');
  my $dbase  = $r->dir_config('DBASE');

  # Balk if we don't have a user to check.
  my $user = $r->user
    or $r->note_basic_auth_failure && return AUTH_REQUIRED;
  foreach my $requires (@{$r->requires}) {
    my ($directive, @list) = split " ", $requires->{requirement};

    # We're ok if only valid-user was required.
    return OK if lc($directive) eq 'valid-user';

    # Likewise if the user requirement was specified and
    # we match based on what we already know.
    return OK if lc($directive) eq 'user' && grep { $_ eq $user } @list;

    # Now for the real work - authorize the user based on Oracle role.
    # This would cover an httpd.conf entry like:
    # Require group DBA
    if ($directive eq 'group') {
      my $dbh = DBI->connect($dbase, $dbuser, $dbpass,
        {RaiseError => 1, PrintError => 1}) or die $DBI::errstr;

      my $sql= qq(
         select grantee
           from dba_role_privs
           where grantee = UPPER(?)
           and   granted_role = UPPER(?)
      );

      my $sth = $dbh->prepare($sql);

      foreach my $role (@list) {
        $sth->execute($r->user, $role);

        my ($ok)  = $sth->fetchrow_array;

        $sth->finish;

        return OK if $ok;
      }
    }
  }

  # No criteria was met so the user didn't pass.
  $r->note_basic_auth_failure;
  return AUTH_REQUIRED;
}
1;
```

Comments

Apache provides one final phase where you can step in and control exactly who has access to your resources. After a user passes authentication, and has successfully proven that he is who he says he is, the authorization phase is entered. This phase is used to further restrict the set of available resources based on actual properties of the known user. The `PerlAuthzHandler` is the hook into this phase, and makes for a convenient addition to your mod_perl arsenal.

In order to take full advantage of the modular design of the Apache access control framework, you need to resist the temptation to write multiple `PerlAuthenHandlers` in order to separate your users into different categories. Remember that the goal of a `PerlAuthenHandler` is merely to make sure that a given username and password match up. Any logic that requires knowledge of the actual user ought to be placed in a `PerlAuthzHandler`. This allows your application to remain seamless for both, say, administrators and view-only users—the administrators, who are granted access to the super-special, eyes-only areas of your site are not required to remember a separate login or to input their credentials a second time based on a different authorization realm.

On the whole, a `PerlAuthzHandler` is pretty much the same as a `PerlAuthenHandler`. Both call `note_basic_auth_failure()` and return `AUTH_REQUIRED` to notify Apache that the user did not have the proper permissions, and return `OK` on success. The main difference is the use of the `requires()` method.

The `requires()` method will return an array reference of hash references. Each hash reference represents a single `Require` directive. For instance, given the directives

```
Require user grier ryan
Require group admiral
```

the result from `requires()` could be represented as

```
[ { requirement => 'user grier ryan',
    method => -1},
  { requirement => 'group admiral',
    method => -1},
];
```

Although the meaning of the `requirement` key is fairly obvious, the `method` key is sitting there looking rather cryptic. As it turns out, `Require` can be governed by the `Limit` directive so that only certain users can, say, `TRACE` or `DELETE`. The `method` key is a bitmask representing which HTTP methods are allowed for a given user. Because it is almost always inappropriate to use the `Limit` core directive, this key can be safely ignored.

Our example code shows one way of iterating through the `requires()` array and dealing with all the possible `Require` configurations. First, we return `OK` if any of the directives are `valid-user`. `valid-user` is a convention used to signify that the user merely has to be authenticated—any user whose credentials can be verified will do. In this case, we want to terminate the authorization process immediately and note a successful return back to Apache, because *some* user was already validated by the authentication phase. Next, the `user` entries are checked. Because it is a requirement for an authentication handler to set the value of `$r->user()`, we can simply verify this value against our list of acceptable users and be on our way.

The only real work that needs to be done is for the `group` list. For this, you can insert any mechanism you want. Here we merely check the object permissions for the validated user within the database itself. Other mechanisms might be similar to those chosen for `PerlAuthenHandlers`. In fact, as with `PerlAuthenHandlers`, a number of actively maintained `PerlAuthzHandlers` are on CPAN, so be sure to consult them before reinventing the authorization wheel.

The interesting thing about the `Require` directive is that the values listed in the Apache documentation (the traditional `user`, `group`, and `valid-user` directives, as well as the newer `file-owner` and `file-group`) are merely conventions; there is no request-time validation of this directive by Apache, despite its implementation within http_core. Modules such as mod_auth are written to work with these standard values, but are intelligent enough to realize that there might be requirements that they cannot satisfy but that other handlers can. As a result, the configuration of the `Require` directive can really be anything that suits your needs. After you know that you can access the raw directive data using `requires()`, it is easy to begin to leverage `Require` to your pre-existing back-end data.

13.7. Writing Your Own Authentication Mechanism

You want to leverage Apache's authentication mechanism but without using the standard browser pop-up boxes.

Technique

Create a custom login form that returns the username and password and use that information to manage the user from a `PerlAuthenHandler`.

```
package Cookbook::CookieAuthentication;

use Apache::Constants qw(OK REDIRECT SERVER_ERROR DECLINED FORBIDDEN);
use Apache::Cookie;
use Apache::Log;
use Apache::Request;

use MIME::Base64 qw(encode_base64 decode_base64);

use strict;

@Cookbook::CookieAuthentication::ISA = qw(Apache::Request);

sub new {

  my ($class, $r) = @_;

  $r = Apache::Request->new($r);

  return bless {r => $r}, $class;
}

sub get_cookie_auth_pw {

  my $r = shift;

  my $log = $r->server->log;

  my $auth_type = $r->auth_type;
  my $auth_name = $r->auth_name;

  # Check that the custom login form was specified.
  my $login = $r->dir_config('Login');

  unless ($login) {
    $log->error("Must specify a login form");
    return SERVER_ERROR;
  }

  $r->custom_response(FORBIDDEN, $login);

  # Check that we're supposed to be handling this.
  unless (lc($auth_type) eq 'cookie') {
    $log->info("AuthType $auth_type not supported by ", ref($r));
    return DECLINED;
  }

  # Check that AuthName was set.
  unless ($auth_name) {
    $log->error("AuthName not set");
    return SERVER_ERROR;
  }

  # Try to get the authentication cookie.
  my %cookiejar = Apache::Cookie->new($r)->parse;

  unless ($cookiejar{$auth_name}) {
    $r->note_cookie_auth_failure;
    return FORBIDDEN;
  }

  # Get the username and password from the cookie.
  my %auth_cookie = $cookiejar{$auth_name}->value;

  my ($user, $password) = split /:/, decode_base64($auth_cookie{Basic}, 2);

  unless ($user && $password) {
    # Whoops, cookie came back without user credentials.

    # Ok, see if we got any credentials from a login form.
    $user = $r->param('user');
    $password = $r->param('password');

    # Don't overwrite the URI in the old cookie, just return.
    return FORBIDDEN unless ($user && $password);

    # We have some credentials, so set an authorization cookie.
    my @values = (uri => $auth_cookie{uri},
                  Basic => encode_base64(join ":", ($user,$password)),
                 );

    $cookiejar{$auth_name}->value(\@values);
    $cookiejar{$auth_name}->path('/');

    $cookiejar{$auth_name}->bake;
    # Now redirect back to where the user was headed
    # and start the cycle again.
    $r->headers_out->set(Location => $auth_cookie{uri});
    return REDIRECT;
  }

  # Ok, we must have received a proper cookie,
  # so pass the info back.
  $r->user($user);
  $r->connection->auth_type($auth_type);

  return (OK, $password);
}

sub note_cookie_auth_failure {

  my $r = shift;

  my $auth_cookie = Apache::Cookie->new($r,
                                        -name => $r->auth_name,
                                        -value => { uri => $r->uri },
                                        -path => '/'
                                       );
  $auth_cookie->bake;
}
1;
```

Comments

One interesting use of a `PerlAuthenHandler` is to create a custom login form that you can present to users to obtain their credentials, rather than the standard, dull pop-up box. Using a custom mechanism has several advantages. The first is that it adds a professional look to your site (imagine that amateurish gray box over the top of your slick, Flash-driven site). The second, and possibly more important, advantage is that by foregoing the standard pop-up, you have the ability to log the user out of the application, which is impossible to do with standard browsers, short of closing the entire browser.

Here we describe one way to implement a custom authentication mechanism using cookies. It is not as full-featured as some of the other modules available on CPAN, like `Apache::AuthCookie`, which was presented in Recipe 10.4. However, it will show how easy it is to implement a custom user authentication scheme that can (almost) transparently replace the standard HTTP authorization methods. As usual, please look to CPAN for more well-tested and proven implementations before attempting to roll your own.

A typical configuration using this new mechanism is really not much different than using conventional Basic authentication. Here is a sample `httpd.conf` configuration.

```
PerlModule Cookbook::CookieAuthentication
PerlModule My::AuthHandler

<Location /private>
  PerlAuthenHandler My::AuthHandler

  PerlSetVar Login "/custom_login.html"

  AuthType Cookie
  AuthName "My Private Documents"
  require valid-user
</Location>
```

The custom login form specified through the `PerlSetVar Login` directive can be as minimal as you want, but must contain the basic requirements our `Cookbook::CookieAuthentication` class expects: setting the `user` and `password` fields, and having an action that points back to the original location governed by the `PerlAuthenHandler` handler. A minimal such form might look like

```
<HTML>
  <HEAD><TITLE>Please log in</TITLE></HEAD>
  <BODY>
    Please login.<BR>
    <FORM METHOD="GET" ENCTYPE="application/x-www-form-urlencoded"
      ACTION="http://www.example.com/private">
      Username: <INPUT TYPE="text" NAME="user"><BR>
      Password: <INPUT TYPE="text" NAME="password"> <BR>
      <INPUT TYPE="submit" VALUE="Log in">
    </FORM>
  </BODY>
</HTML>
```

but it is easy to see how this could be made into a template that substituted the form action with `$r->location()` on-the-fly using a `PerlHandler`. The only point to note is that the login form itself cannot be protected—if it were, the user could never see the form in order to use it.

If you thought that the Basic authentication challenge/response cycle was complex, we are sorry to report that implementing a custom scheme does not simplify matters. In fact, the redirection and cookie setting presented here adds an extra layer of complexity, which is probably another argument for using a module maintained on CPAN. Nevertheless, stepping through the code here is an interesting and fruitful exercise, because it demonstrates many of the concepts previously explored.

Our `Cookbook::CookieAuthentication` example module begins by subclassing the `Apache::Request` class in the same way as presented in Recipe 10.8. This allows us to do two important things. First, we can add our new methods, `get_cookie_auth_pw()` and `note_cookie_auth_failure()` directly to `$r`, which makes them fit neatly into a structure that the end programmer already understands. More interestingly, however, is that now the Apache request object that gets passed to our methods inherits from the `Apache::Request` class and not the `Apache` class. This means we can use `$r->param()` from within our own methods without needing to call `Apache::Request->new()` ourselves.

After defining our constructor we proceed to the trigger that actually makes the entire process work. Using the `custom_response()` method we set the custom login form to intercept any `FORBIDDEN` server responses with our custom response. Basically, any time our module decides that the user has not supplied sufficient information, the login form is presented in an entirely transparent manner. Unlike with traditional authentication, we cannot merely override an `AUTH_REQUIRED` response with our custom response because the client would answer the resulting `401 Authorization Required` response with the standard pop-up box, which is what we are trying to avoid.

The remainder of the code defines our two main methods, which perform functions analogous to the `get_basic_auth_pw()` and `note_basic_auth_failure()` methods upon which they are based. `note_cookie_auth_failure()` presents the client with a challenge in a manner similar to Basic authentication. In this case, however, the challenge is in the form of a cookie, which will be sent along with any future requests to the server. This is used to track the initial request URL so that we can redirect to the original destination later.

The net result of the `get_cookie_auth_pw()` method is exactly the same as its cousin, `get_basic_auth_pw()`. It returns either `OK` and the gleaned password or a status code to be propagated back to Apache. How it accomplishes this is rather complex. After first checking that we are supposed to be handling this request, the method looks for a cookie with a name of the realm specified by the `AuthName` directive. If this is not present, a call is made to `$r->note_cookie_auth_failure()`, which initializes the challenge, and a `FORBIDDEN` code is returned, which silently presents the user with the custom login form.

That completes the initial request, which is mechanically different from that of Basic authentication but has served the same purpose: The end-user is presented with a way to enter some credentials. After the credentials are supplied, a second request is made, which our `PerlAuthHandler` again traps, and which again gets caught by the `get_cookie_auth_pw()` method. This time, there are some credentials to extract and place within the cookie in the form of a base64-encoded `user:passwd` string. It is here that we depart from the regularly scheduled program.

At this point, we could just set the username and return the password but there are two problems that need to be overcome. First, the target action of our form was not the requested URL, so the end user will not end up where he was originally headed. Additionally, we have no verification that the cookie was accepted, so passing this request along would mean that the next request would again be redirected to the login form. The solution is to cut this second request short and redirect back to the original URL, where the credentials can be checked directly from the cookie.

If the credentials can be gleaned from the cookie, then the challenge is considered met and `get_cookie_auth_pw()` sets `$r->user()` and `$r->connection->auth_type()`. This is a requirement of all authentication handlers, and is more fully discussed in Recipe 13.8. `get_cookie_auth_pw()` then returns the decoded password for authentication by the invoking `PerlAuthenHandler`, along with an `OK` status. Assuming this authentication is successful, subsequent calls to documents within the specified realm will then use the user and password information directly in the cookie, in effect treating it as an `Authorization` header.

Although `CookieAuthentication.pm` was rather long, by subclassing `Apache::Request` its use in a `PerlAuthenHandler` handler is rather easy and looks remarkably similar to the `PerlAuthenHandler` presented in Recipe 13.3 that used the standard Basic authentication scheme.

```
package My::AuthHandler;

use Apache::Constants qw(OK FORBIDDEN);

use Cookbook::CookieAuthentication;
use My::Utils qw(authenticate_user);

use strict;

sub handler  {

  my $r = Cookbook::CookieAuthentication->new(shift);
```

```
    # Let subrequests pass.
    return OK unless $r->is_initial_req;

    # Get the client-supplied credentials.
    my ($status, $password) = $r->get_cookie_auth_pw;

    return $status unless $status == OK;

    # Perform some custom user/password validation.
    return OK if authenticate_user($r->user, $password);

    # Whoops, bad credentials.
    $r->note_cookie_auth_failure;
    return FORBIDDEN;
  }
  1;
```

There are only a few minor differences here to be aware of when using our new custom authentication methods. The request object is retrieved using our constructor method `Cookbook::CookieAuthentication->new()`, and the core mod_perl methods are replaced by the new methods supplied by `CookieAuthentication.pm`. Additionally, the proper return value to follow `$r->note_cookie_auth_failure()` is `FORBIDDEN` instead of `AUTH_REQUIRED`. Despite these differences, it should be easy to see that we have effectively leveraged some of the object-oriented techniques presented earlier in order to almost transparently substitute an authentication scheme built in to the `HTTP/1.1` protocol with our own, entirely different scheme.

13.8. Using Digest Authentication

You want to use the more secure Digest authentication mechanism but want to store the user data in something other than a flat file.

Technique

Use the following API, based on mod_digest, in a custom `PerlAuthenHandler`.

```
package Cookbook::DigestAPI;

use Apache;
use Apache::Constants qw(OK DECLINED SERVER_ERROR AUTH_REQUIRED);
use 5.006;
use Digest::MD5;
use DynaLoader;

use strict;

our @ISA = qw(DynaLoader Apache);
our $VERSION = '0.01';

__PACKAGE__->bootstrap($VERSION);

sub new {

  my ($class, $r) = @_;

  $r ||= Apache->request;

  return bless { r => $r }, $class;
}

sub get_digest_auth_response {

  my $r = shift;

  return DECLINED unless lc($r->auth_type) eq 'digest';

  return SERVER_ERROR unless $r->auth_name;

  # Get the response to the Digest challenge.
  my $auth_header = $r->headers_in->get($r->proxyreq ?
                                        'Proxy-Authorization' :
                                        'Authorization');

  # We issued a Digest challenge - make sure we got Digest back.
  $r->note_digest_auth_failure && return AUTH_REQUIRED
    unless $auth_header =~ m/^Digest/;

  # Parse the response header into a hash.
  $auth_header =~ s/^Digest\s+//;
  $auth_header =~ s/"//g;

  my %response = map { split(/=/) } split(/,\s*/, $auth_header);
  # Make sure that the response contained all the right info.
  foreach my $key (qw(username realm nonce uri response)) {
    $r->note_digest_auth_failure && return AUTH_REQUIRED
      unless $response{$key};
  }

  # Ok, we're good to go. Set some info for the request
  # and return the response information so it can be checked.
  $r->user($response{username});
  $r->connection->auth_type('Digest');

  return (OK, \%response);
}

sub compare_digest_response {
  # Compare a response hash from get_digest_auth_response()
  # against a pre-calculated digest (e.g., a3165385201a7ba52a12e88cb606bc76).

  my ($r, $response, $digest) = @_;

  my $md5 = Digest::MD5->new;

  $md5->add(join ":", ($r->method, $response->{uri}));

  $md5->add(join ":", ($digest, $response->{nonce}, $md5->hexdigest));

  return $response->{response} eq $md5->hexdigest;
}
1;
```

Comments

Digest authentication is similar to Basic authentication in most respects. It follows the same challenge/response mechanism discussed in Recipe 13.3, even using the same set of headers. As with Basic authentication, the server responds with `401 Authorization Required` and the `WWW-Authenticate` header whenever client credentials have not met the server challenge. Likewise, to communicate with the server, the client passes its credentials along in the form of an `Authorization` request header. The main difference between the two authentication schemes is that with Digest authentication the password entered by the end user is never transmitted over the connection. Instead, the client response is a combination of several distinct bits of information in clear-text, along with an MD5 hash derived from the same information plus the password.

The MD5 hashing algorithm is a one-way hash function—it takes input data and returns a fixed-length hash which can then act as a unique fingerprint for the data. In reality, a generated MD5 hash is *not* unique across all possible datasets, but one of the properties of a one-way hash is that it is *collision-resistant*. In practical terms, this means that would-be attackers must resort to a brute-force methodology to undermine the security of the data. See Chapter 18 in Bruce Schneier's *Applied Cryptography* for a more detailed discussion of one-way hashes specifics of the MD5 algorithm.

This MD5 fingerprint, and the fact that it is practically impossible to derive the underlying data from the hash itself, provide the core concepts for the Digest authentication scheme. The idea here is that if both parties (the client and server) have access to the same information, then both ought to be able to create the same, unique MD5 hash and there is no reason for either side to transmit the actual password across the wire.

The comparison of information begins on the server. Here, the username, password, and realm are hashed together using the MD5 algorithm and the resulting hash is stored for later use. Typically, the hash is generated by the `htdigest` utility (which is automatically installed in your Apache installation tree) and placed into a file on disk. A sample entry generated by `htdigest` might look like

```
authors:cookbook:8901089be1ee922e5d6d2193f9ef620a
```

where the first value is the user, followed by the realm and the MD5 hash just described.

When a request comes in for a protected document, the server begins the cycle by transmitting a `WWW-Authenticate` header with the authentication scheme, realm, and a server generated *nonce* value. The nonce is unique to the current authentication session and, if properly implemented, can be used to ensure that the session is fresh and not a replay of an old session caught through network eavesdropping.

On the client side, the username and password are entered by the end user based on the authentication realm sent from the server. This is typically accomplished using the same pop-up box we saw in the Basic authentication scheme.

At this point, the client has access to the same information sent to the `htdigest` utility and can produce a matching MD5 hash. However, rather than transmit this hash back to the server, the client creates a new hash based on the password hash and additional information from the request, such as the server-generated nonce and the request URI. This new hash is sent back to the server via the `Authorization` header, along with the nonce and other data about the request.

Now, because the client and server have had (at various points in time) access to the same dataset—the user-supplied username and password, as well as the request URI, authentication realm, and other information shared in the HTTP headers—both ought to be able to generate the same MD5 hash. If the hashes do not agree, the difference can be attributed to the one piece of information not mutually agreed upon through the HTTP request: the password.

A typical dialogue might look similar to the following output. Note the `Authorization` header, which is a bit more complex than that seen during a Basic authentication session, as well as the inclusion of the `nonce` attribute passed with the `WWW-Authenticate` header.

```
GET /index.html HTTP/1.1
Accept: text/html, image/png, image/jpeg, image/gif, image/x-xbitmap, */*
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Accept-Language: en
Cache-Control: no-cache
Connection: Keep-Alive, TE
Host: jib.example.com
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 2000) Opera 5.12

HTTP/1.1 401 Authorization Required
WWW-Authenticate: Digest realm="cookbook", nonce="1003585655"
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

GET /index.html HTTP/1.1
Accept: text/html, image/png, image/jpeg, image/gif, image/x-xbitmap, */*
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Accept-Language: en
Authorization: Digest username="authors", realm="cookbook", uri="/index.html",
algorithm=MD5, nonce="1003585655", response="48835a6cc022661cb365da39eeba068e"
Cache-Control: no-cache
Connection: Keep-Alive, TE
Host: jib.example.com
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 2000) Opera 5.12

HTTP/1.1 200 OK
Last-Modified: Sat, 06 Oct 2001 14:16:34 GMT
ETag: "4987-51e-3bbf1242"
Accept-Ranges: bytes
```

```
Content-Length: 1310
Keep-Alive: timeout=15, max=99
Connection: Keep-Alive
Content-Type: text/html
```

What we have just presented is merely a high-level sketch of the overall process; for more detailed coverage see RFC 2617. It should be apparent, however, that Digest authentication offers a significant security improvement over Basic authentication. Not only is the password safe from packet sniffers and other such malicious devices, but no record of the actual password exists on the server, which also adds another level of security from scorned ex-employees. The main downside of this approach is that the actual password is not important once the hash has been generated—whereas with Basic authentication the password is encrypted on the server and knowing the encrypted value is of little use; with Digest authentication knowing only the hash is sufficient to gain access to server resources.

Despite this shortcoming, keeping passwords from being transmitted in the open is a definite improvement. Although it used to be the case that browser support for Digest authentication was minimal, the trend seems to be shifting toward accepting this more secure form. Current versions of Microsoft Internet Explorer, Opera, Konqueror, and Amaya all support both Digest and Basic authentication. On the Apache side, both mod_digest and mod_auth_digest implement the Digest authentication protocol. mod_digest is the older of the two, whereas mod_auth_digest is relatively new and is relegated to the experimental module region of the Apache distribution. Both modules hold user credentials in a file specified by the `AuthDigestFile` directive, which currently is the only source available for storing user information on the server side.

The example code in this recipe tries to provide programmers wanting to implement Digest authentication the same flexibility granted to the Basic authentication scheme. `Cookbook::DigestAPI` is an API for the Digest authentication scheme based on mod_digest. Before we begin, we need to note that by modeling the code after mod_digest, we have introduced a major drawback—only mod_auth_digest provides the full Digest functionality expected by Internet Explorer, which is arguably the most popular browser capable of using Digest authentication. Unfortunately, the code for mod_auth_digest is significantly more complex than that of mod_digest and does not lend itself to an example easily. Thus, we chose the mod_digest implementation for clarity, as well as the fact that it is supported by all the other browsers in our list.

So, with the usual caveats out of the way, we can proceed. The `Cookbook::DigestAPI` class offers an API analogous to the `get_basic_auth_pw()` and `note_basic_auth_failure()` methods in the `Apache` class. It was designed as a subclass of `Apache`, using the techniques described in Recipe 10.5, so all a `PerlAuthenHandler` needs to do is grab the Apache request object using the `new()` constructor from our class instead of from `Apache->request()`. This will add the new API directly to `$r`, making programming with `Digest` authentication as simple and flexible as possible.

The use of `DynaLoader` in the solution code, `DigestAPI.pm`, should have clued you in that the code is actually incomplete as shown. As it turns out, the Apache API already provides part of the API we will need via the core `ap_note_digest_auth_failure` function; it just is not carried over to us through mod_perl. Rather than code the functionality ourselves, we chose to reuse the existing Apache API by adding a bit of XS code to the mix. Here is `DigestAPI.xs`.

Listing 13.3 `DigestAPI.xs`

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include "mod_perl.h"

MODULE = Cookbook::DigestAPI         PACKAGE = Cookbook::DigestAPI

PROTOTYPES: ENABLE

void
note_digest_auth_failure(r)
  Apache r

  CODE:
    ap_note_digest_auth_failure(r);
```

Because we are using XS, we need to bring together `DigestAPI.pm` and `DigestAPI.xs` with a `Makefile.PL`.

Listing 13.4 `Makefile.PL` *for* `Cookbook::DigestAPI`

```
#!perl

use ExtUtils::MakeMaker;
use Apache::src ();
use Config;

use strict;

my %config;

$config{INC} = Apache::src->new->inc;
```

Listing 13.4 *(continued)*

```
if ($^O =~ /Win32/) {
  require Apache::MyConfig;

  $config{DEFINE}  = ' -D_WINSOCK2API_ -D_MSWSOCK_ ';
  $config{DEFINE} .= ' -D_INC_SIGNAL -D_INC_MALLOC '
    if $Config{usemultiplicity};

  $config{LIBS} =
    qq{ -L"$Apache::MyConfig::Setup{APACHE_LIB}" -lApacheCore } .
    qq{ -L"$Apache::MyConfig::Setup{MODPERL_LIB}" -lmod_perl};
}

WriteMakefile(
  NAME         => 'Cookbook::DigestAPI',
  VERSION_FROM => 'DigestAPI.pm',
  PREREQ_PM    => { mod_perl => 1.26 },
  ABSTRACT     => 'An XS-based Apache module',
  AUTHOR       => 'authors@modperlcookbook.org',
  %config,
);
```

We also need a `typemap` file to translate our Apache request object into a format the Apache API can understand.

Listing 13.5 `typemap` *for* `Cookbook::DigestAPI`

```
TYPEMAP
Apache  T_APACHEOBJ

OUTPUT
T_APACHEOBJ
  sv_setref_pv($arg, \"${ntype}\", (void*)$var);

INPUT
T_APACHEOBJ
  r = sv2request_rec($arg, \"$ntype\", cv)
```

With the exception of the module name, these two files are no different than the files from Recipe 3.19, so you can look to that recipe for guidance on using `h2xs` to create a skeleton XS framework.

The resulting `note_digest_auth_failure()` method provided by `DigestAPI.xs` does the same thing as its `note_basic_auth_failure()` counterpart, taking the necessary steps to ensure that the client receives a challenge with the appropriate information. The other half of our API is handled in Perl and shown in the solution module DigestAPI.pm.

The implementation of get_digest_auth_response() is fairly straightforward. First, the method goes through a few basic checks, like making sure that the AuthType setting is correct and that an authorization realm is specified. Next it grabs the incoming Authorization header and parses it into its constituent parts. In order to ensure that we can generate a proper MD5 hash when required, we need to make certain that the header contains at least a few specific fields.

After all our checks are complete, we set the username and authentication type for the request so that later handlers can access this information and make decisions around it. You should note that we set $r->connection->auth_type() and not $r->auth_type(). As it turns out, these two methods, while similar in name, are dealing with fundamentally different details about the request: $r->connection->auth_type() is really a slot in the Apache connection record that represents the authentication type agreed upon by the client and server at request time, whereas $r->auth_type() corresponds to the AuthType directive set in httpd.conf. For the curious, mod_perl actually digs out the AuthType setting of the per-directory configuration for http_core using a mechanism similar to that described in Recipe 8.14.

If you find it puzzling that authentication information is stored on a per-connection basis (which can involve multiple requests), the reasons are mainly historical and go back to when Apache was a single digit release. But fear not, $r->connection->auth_type() is wiped clean at the end of each request.

Unlike with get_basic_auth_pw(), where we only needed a status and password to check, for Digest authentication a number of values are needed from the incoming Authorization header in order to create the MD5 hash and authenticate the user. So, the get_digest_auth_response() method returns a hash reference on success instead of a plain-text password.

Because comparing the encrypted user credentials stored on the server to the information passed by the client is a complex and tedious task, our new API provides a method to ease the pain. compare_digest_response() takes the response hash returned by get_digest_auth_response() and an MD5 hash of user credentials and compares them, returning true if they match and shielding the application programmer from the technicalities.

The result is a nice, tidy subclass of Apache that conveniently adds the methods we will need to process Digest authentication independent of the AuthDigestFile directive. Here is an example configuration:

```
PerlModule Cookbook::DigestAPI
PerlModule Cookbook::AuthDigestDBI

<Location /private>
  PerlAuthenHandler Cookbook::AuthDigestDBI

  AuthType Digest
  AuthName "cookbook"
  Require valid-user
</Location>
```

Both the configuration and use of the API are nearly identical to what you might find in a `PerlAuthenHandler` using Basic authentication. Here is a sample handler that puts our new class to use.

```
package Cookbook::AuthDigestDBI;

use Apache::Constants qw(OK DECLINED AUTH_REQUIRED);

use Cookbook::DigestAPI;
use DBI;
use DBD::Oracle;

use strict;

sub handler {

  my $r = Cookbook::DigestAPI->new(shift);

  return DECLINED unless $r->is_initial_req;

  my ($status, $response) = $r->get_digest_auth_response;

  return $status unless $status == OK;

  my $user  = $r->dir_config('DBUSER');
  my $pass  = $r->dir_config('DBPASS');
  my $dbase = $r->dir_config('DBASE');

  my $dbh = DBI->connect($dbase, $user, $pass,
   {RaiseError => 1, AutoCommit => 1, PrintError => 1}) or die $DBI::errstr;
  my $sql= qq(
     select digest
       from user_digests
       where username = ?
       and   realm = ?
  );

  my $sth = $dbh->prepare($sql);

  $sth->execute($r->user, $r->auth_name);

  my ($digest) = $sth->fetchrow_array;

  $sth->finish;

  return OK if $r->compare_digest_response($response, $digest);

  $r->note_digest_auth_failure;
  return AUTH_REQUIRED;
}
1;
```

As we mentioned, this example follows an older version of the Digest scheme and is not entirely compatible with even the browsers that claim to support Digest authentication. However, it does provide a framework for a complete solution, either as a starting point for a new implementation or as a class that can be extended using object-oriented techniques.