```
  # 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.