Integrating mod_perl with Apache 2.1 Authentication

By Geoffrey Young on July 8, 2003 12:00 AM

Scratching Your Own Itch

Some time ago I became intrigued with Digest authentication, which uses the same general mechanism as the familiar Basic authentication scheme but offers significantly more password security without requiring an SSL connection. At the time it was really just an academic interest—while some browsers supported Digest authentication, many of the more popular ones did not. Furthermore, even though the standard Apache distribution came with modules to support both Basic and Digest authentication, Apache (and thus mod_perl) only offered an API for interacting with Basic authentication. If you wanted to use Digest authentication, flat files were the only password storage medium available. With both of these restrictions, it seemed impractical to deploy Digest authentication in all but the most limited circumstances.

Fast forward two years. Practically all mainstream browsers now support Digest authentication, and my interest spawned what is now Apache::AuthDigest, a module that gives mod_perl 1.0 developers an API for Digest authentication that is very similar to the Basic API that mod_perl natively supports. The one lingering problem is probably not surprising—Microsoft Internet Explorer. As it turns out, using the Digest scheme with MSIE requires a fully RFC-compliant Digest implementation, and Apache::AuthDigest was patterned after Apache 1.3's `mod_digest.c`, which is sufficient for most browsers but not MSIE.

In my mind, opening up Digest authentication through mod_perl still needed work to be truly useful, namely full RFC compliance to support MSIE. Wading through RFCs is not how I like to spend my spare time, so I started searching for a shortcut. Because Apache 2.0 did away with `mod_digest.c` and replaced it with the fully compliant `mod_auth_digest.c`, I was convinced that there was something in Apache 2.0 I could use to make my life easier. In Apache 2.1, the development version of the next generation Apache server, I found what I was looking for.

In this article, we're going to examine a mod_perl module that provides Perl support for the new authentication provider hooks in Apache 2.1. These authentication providers make writing Basic authentication handlers easier than it has been in the past. At the same time, the new provider mechanism opens up Digest authentication to the masses, making the Digest scheme a real possibility for filling your dynamic authentication needs. While the material is somewhat dense, the techniques we will be looking at are some of the most interesting and powerful in the mod_perl arsenal. Buckle up.

To follow along with the code in this article, you will need at least mod_perl version 1.99_10, which is currently only available from CVS. You will also need Apache 2.1, which is also only available from CVS. Instructions for obtaining the sources for both can be found here. When compiling Apache, keep in mind that the code presented here only works under the prefork MPM — making it thread-safe is the next step in the adventure.

Authentication Basics

Because there is lots of material to cover, we'll skip over the requisite introductory discussion of HTTP authentication, the Apache request cycle, and other materials that probably already familiar and skip right to the mod_perl authentication API. In both mod_perl 1.0 and mod_perl 2.0, the `PerlAuthenHandler` represents Perl access to the Apache authentication phase, where incoming user credentials are traditionally matched to those stored within the application. A simple `PerlAuthenHandler` in mod_perl 2.0 might look like the following.

```
package My::BasicHandler;

use Apache::RequestRec ();
use Apache::Access ();

use Apache::Const -compile => qw(OK DECLINED HTTP_UNAUTHORIZED);

use strict;

sub handler {
  my $r = shift;

  # get the client-supplied credentials
  my ($status, $password) = $r->get_basic_auth_pw;

  # only continue if Apache says everything is OK
  return $status unless $status == Apache::OK;

  # user1/basic1 is ok
  if ($r->user eq 'user1' && $password eq 'basic1') {
    return Apache::OK;
  }

  # user2 is denied outright
  if ($r->user eq 'user2') {
    $r->note_basic_auth_failure;
    return Apache::HTTP_UNAUTHORIZED;
  }

  # all others are passed along to the Apache default
  # handler, which reads from the AuthUserFile
  return Apache::DECLINED;
}

1;
```

Although simple and impractical, this handler illustrates the API nicely. The process begins with a call to `get_basic_auth_pw()`, which does a few things behind the scenes. If a suitable Basic Authorization header is found, `get_basic_auth_pw()` will parse and decode the header, populate the user slot of the request record, and return `OK` along with the user-supplied password in clear text. Any value other than `OK` should be immediately propagated back to Apache, which effectively terminates the current request.

The next step in the process is where the real authentication logic resides. Our handler is responsible for digging out the username from `$r->user()` and applying some criteria for determining whether the user-supplied credentials are acceptable. If they are, the handler simply returns `OK` and the request is allowed to proceed. If they are not, the handler has a decision to make: either call `note_basic_auth_failure()` and return `HTTP_UNAUTHORIZED` (which is the same as the old `AUTH_REQUIRED`) to indicate failure, or return `DECLINED` to pass authentication control to the next authentication handler.

For the most part, the mod_perl API is identical to the API Apache offers to C module developers. The benefit that mod_perl adds is the ability to easily extend authentication beyond Apache's default flat-file mechanism to the areas where Perl support is strong, such as relational databases or LDAP. However, despite the versatility and strength programming the authentication phase offered, I never liked the look and feel of the API. While in some respects the process is dictated by the nuances of RFC 2617 and the HTTP protocol itself, the interface always struck me as somewhat inconsistent and difficult for new users to grasp. Additionally, as already mentioned, the API covers only Basic authentication, which is a real drawback as more and more browsers support the Digest scheme.

Apparently I wasn't alone in some of these feelings. Apache 2.1 has taken steps to improve the overall process for module developers. The result is a new, streamlined API that focuses on a new concept: authentication providers.

Authentication Providers in Apache 2.1

While in Apache 2.0 module writers were responsible for a large portion of the authentication logic—calling routines to parse and set authentication headers, digging out the user from the request record, and so on — the new authentication mechanism in Apache 2.1 delegates all HTTP and RFC logic out to two standard modules. `mod_auth_basic` handles Basic authentication and is enabled in the default Apache build. The standard `mod_auth_digest`, not enabled by default, handles the very complex world of Digest authentication. Regardless of the authentication scheme you choose to support, these modules are responsible for the details of parsing and interpreting the incoming request headers, as well as generating properly formatted response headers.

Of course, managing authentication on an HTTP level is only part of the story. What `mod_auth_basic` and `mod_auth_digest` leave behind is the job of digging out the server-side credentials and matching them to their incoming counterpart. Enter authentication providers.

Authentication providers are modules that supply server-side credential services to `mod_auth_basic` or `mod_auth_digest`. For instance, the default `mod_authn_file` digs the username and password out of the flat file specified by the `AuthUserFile` directive, similar to the default mechanism in Apache 1.3 and 2.0. An Apache 2.1 configuration that explicitly provides the same flat file behavior as Apache 2.0 would look similar to the following.

```
<Location /protected>
  Require valid-user
  AuthType Basic
  AuthName realm1

  AuthBasicProvider file

  AuthUserFile realm1
</Location>
```

The new part of this configuration is the `AuthBasicProvider` directive, which is implemented by `mod_auth_basic` and used to specify the provider responsible for managing server-side credentials. There is also a corresponding `AuthDigestProvider` directive if you have `mod_auth_digest` installed.

While it could seem as though Apache 2.1 is merely adding another directive to achieve essentially the same results, the shift to authentication providers adds significant value for module developers: a new API that is far simpler than before. Skipping ahead to the punch line, programming with new Perl API for Basic authentication, which follows the Apache API almost exactly, would look similar to the following.

```
package My::BasicProvider;

use Apache::Const -compile => qw(OK DECLINED HTTP_UNAUTHORIZED);

use strict;

sub handler {
  my ($r, $user, $password) = @_;
  # user1/basic1 is ok
  if ($user eq 'user1' && $password eq 'basic1') {
    return Apache::OK;
  }

  # user2 is denied outright
  if ($user eq 'user2') {
    return Apache::HTTP_UNAUTHORIZED;
  }

  # all others are passed along to the next provider
  return Apache::DECLINED;
}

1;
```

As you can see, not only are the incoming username and password supplied in the argument list, removing the need for `get_basic_auth_pw()` and its associated checks, but gone is the need to call `note_basic_auth_failure()` before returning `HTTP_UNAUTHORIZED`. In essence, all that module writers need to be concerned with is validating the user credentials against whatever back-end datastore they choose. All in all, the API is a definite improvement. To add even more excitement, the API for Digest authentication looks almost exactly the same (but more on that later).

Because the new authentication provider approach represents a significant change in the way Apache handles authentication internally, it is not part of the stable Apache 2.0 tree and is instead being tested in the development tree. Unfortunately, until the provider mechanism is backported to Apache 2.0, or an official Apache 2.2 release, it is unlikely that authentication providers will be supported by core mod_perl 2.0. However, this does not mean that mod_perl developers are out of luck—by coupling mod_perl's native directive handler API with a bit of XS, we can open up the new Apache provider API to Perl with ease. The Apache::AuthenHook module does exactly that.

Introducing Apache::AuthenHook

Over in the Apache C API, authentication providers have a few jobs to do: they must register themselves by name as a provider while supplying a callback interface for the schemes they wish to support (Basic, Digest, or both). In order to open up the provider API to Perl modules our gateway module Apache::AuthenHook will need to accomplish these tasks as well. Both of these are accomplished at the same time through a call to the official Apache API function `ap_register_provider`.

Usually, mod_perl provides direct access to the Apache C API for us. For instance, a Perl call to `$r->get_basic_auth_pw()` is proxied off to `ap_get_basic_auth_pw`—but in this case `ap_register_provider` only exists in Apache 2.1 and, thus, is not supported by mod_perl 2.0. Therefore, part of what Apache::AuthenHook needs to do is open up this API to Perl. One of the great things about mod_perl is the ease at which it allows itself to be extended even beyond its own core functionality. Opening up the Apache API past what mod_perl allows is relatively easy with a dash of XS.

Our module opens with `AuthenHook.xs`, which is used to expose `ap_register_provider` through the Perl function `Apache::AuthenHook::register_provider()`.

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

#include "mod_perl.h"
#include "ap_provider.h"
#include "mod_auth.h"

...

static const authn_provider authn_AAH_provider =
{
  &check_password,
  &get_realm_hash,
};

MODULE = Apache::AuthenHook    PACKAGE = Apache::AuthenHook

PROTOTYPES: DISABLE

void
register_provider(provider)
  SV *provider

  CODE:

    ap_register_provider(modperl_global_get_pconf(),
                         AUTHN_PROVIDER_GROUP,
                         SvPV_nolen(newSVsv(provider)), "0",
                         &authn_AAH_provider);
```

Let's start at the top. Any XS module you write will include the first three header files, while any mod_perl XS extension will require at least `#include "mod_perl.h"`. The remaining two included header files are specific to what we are trying to accomplish—`ap_provider.h` defines the `ap_register_provider` function, while `mod_auth.h` defines the `AUTHN_PROVIDER_GROUP` constant we will be using, as well as the `authn_provider` struct that holds our callbacks.

Skipping down a bit, we can see our implementation of `Apache::AuthenHook::register_provider()`. The `MODULE` and `PACKAGE` declarations place `register_provider()` into the `Apache::AuthenHook` package. Following that is the definition of the `register_provider()` function itself.

As you can see, `register_provider()` accepts a single argument, a Perl scalar representing the name of the provider to register, making its usage something akin to the following.

```
Apache::AuthenHook::register_provider('My::BasicProvider');
```

The user-supplied name is then used in the call to `ap_register_provider` from the Apache C API to register `My::BasicProvider` as an authentication provider.

The twist in the process is that our implementation of `ap_register_provider` registers Apache::AuthenHook's callbacks (the `check_password` and `get_realm_hash` routines not shown here) for each Perl provider. In essence, this means that Apache::AuthenHook will be acting as go-between for Perl providers. Much in the same way that mod_perl proper is called by Apache for each phase and dispatches to different `Perl*Handlers`, Apache::AuthenHook will be called by Apache's authentication modules and dispatch to the appropriate Perl provider at runtime.

If this boggles your mind a bit, not to worry, it is only being presented to give you a feel for the bigger picture and show how easy it is to open up closed parts of the Apache C API with mod_perl and just a few lines of XS. However, the fun part of Apache::AuthenHook (and the part that you are more likely to use in your own mod_perl modules) is handled over in Perl space.

Setting the Stage

Now that we have the ability to call `ap_register_provider`, we need to link that into the Apache configuration process somehow. What we do not want to do is replace current `PerlAuthenHandler` functionality, since that directive is for inserting authentication handler logic in place of Apache's defaults. In our case, we need the default modules to run so they can call our Perl providers. Instead, we want to make it possible for Perl modules to register themselves as authentication providers. While we could have Perl providers call our new `register_provider()` function directly, Apache::AuthenHook chose to make the process transparent, using mod_perl's directive handler API to call `register_provider()` silently as `httpd.conf` is parsed.

Apache::AuthenHook makes sneaky use of directive handlers to extend the default Apache `AuthBasicProvider` and `AuthDigestProvider` directives so they register Perl providers on-the-fly.

The net result is that Perl providers will be fully registered and configured via standard Apache directives, similar to the following.

```
AuthBasicProvider My::BasicProvider file
```

At configuration time, Apache::AuthenHook will intercept `AuthBasicProvider` and register My::BasicProvider. At request time, `mod_auth_basic` will attempt to authenticate the user, first using My::BasicProvider, followed by the default file provider if My::BasicProvider declines the request.

A nice side effect to this is that through our implementation we will be giving mod_perl developers a feature they have never had before—the ability to interlace Perl handlers and C handlers within the same phase.

```
AuthDigestProvider My::DigestProvider file My::OtherDigestProvider
```

Exciting, no? Let's take a look at `AuthenHook.pm` and see how the directive handler API works in mod_perl 2.0.

Directive Handlers with mod_perl

Directive handlers are a very powerful but little used feature of mod_perl. For the most part, their lack of use probably stems from the complex and intimidating API in mod_perl 1.0. However, in mod_perl 2.0, the API is much simpler and should lend itself to adoption by a larger audience.

The directive handler API allows mod_perl modules to define their own custom configuration directives that are understood by Apache, For example, enabling modules to make use of configuration variables like:

```
Foo "bar"
```

in `httpd.conf` requires only a few relatively simple settings you can code directly in Perl.

While using directive handlers simply to replace `PerlSetVar` behavior might seem a bit flashy, the techniques used by Apache::AuthenHook are some of the most powerful mod_perl has to offer.

As previously mentioned, we will be extending the new `AuthBasicProvider` and `AuthDigestProvider` directives to apply to Perl providers as well, silently registering each provider as the directive itself is parsed. To do this, we redefine these core directives, manipulate their configuration data, then disappear and allow Apache to handle the directives as if we were never there.

The code responsible for this is in `AuthenHook.pm`.

```
package Apache::AuthenHook;
```

```
use 5.008;

use DynaLoader ();

use mod_perl 1.99_10;     # DECLINE_CMD and $parms->info support
use Apache::CmdParms ();  # $parms->info

use Apache::Const -compile => qw(OK DECLINE_CMD OR_AUTHCFG RAW_ARGS);

use strict;

our @ISA     = qw(DynaLoader);
our $VERSION = '2.00_01';

__PACKAGE__->bootstrap($VERSION);

our @APACHE_MODULE_COMMANDS = (
  { name         => 'AuthDigestProvider',
    errmsg       => 'specify the auth providers for a directory or location',
    args_how     => Apache::RAW_ARGS,
    req_override => Apache::OR_AUTHCFG,
    cmd_data     => 'digest' },

  { name         => 'AuthBasicProvider',
    errmsg       => 'specify the auth providers for a directory or location',
    args_how     => Apache::RAW_ARGS,
    req_override => Apache::OR_AUTHCFG,
    func         => 'AuthDigestProvider',
    cmd_data     => 'basic' },
);
```

At the top of our module we import a few required items, some that are new and some that should already be familiar. DynaLoader and the `bootstrap()` method are required to pull in the `register_provider()` function from our XS implementation and, unlike with mod_perl 1.0, have nothing to do with the actual directive handler implementation. The Apache::CmdParms class provides the `info()` method we will be illustrating shortly, while Apache::Const gives us access to the constants we will need throughout the process.

The `@APACHE_MODULE_COMMANDS` array is where the real interface for directive handlers begins. `@APACHE_MODULE_COMMANDS` holds an array of hashes, each of which defines the behavior of an Apache directive. Let's focus on the first directive our handler implements, `AuthDigestProvider`, forgetting for the moment that mod_auth_digest also defines this directive.

While it should be obvious that the `name` key specifies the name of the directive, it is not so obvious that it also specifies the default Perl subroutine to call when Apache encounters `AuthDigestProvider` while parsing `httpd.conf`. Later on, we will need to implement the `AuthDigestProvider()` subroutine, which will contain the logic for all the activities we want to perform when Apache sees the `AuthDigestProvider` directive.

The `args_how` and `req_override` are fields that tell Apache specifically how the directive is supposed to behave in the configuration. `req_override` defines how our directive will interact with the core `AllowOverride` directive, in our case allowing `AuthDigestProvider` in `.htaccess` files only on directories governed by `AllowOverride AuthConfig`. Similarly, `args_how` defines how Apache should interact with our `AuthDigestProvider()` subroutine when it sees our directive in `httpd.conf`. In the case of `RAW_ARGS`, it means that Apache will pass our callback whatever follows the directive as a single string. Other possible values for both of these keys can be found in the documentation pointed to at the end of this article.

The final important key in our first hash is the `cmd_data` key, in which we can store a string of our choosing. This will become important in a moment.

The second hash in `@APACHE_MODULE_COMMANDS` defines the behavior of the `AuthBasicProvider` directive, which for the most part is identical to `AuthDigestProvider`. The differences are important, however, and begin with the addition of the `func` key. Although the default Perl subroutine callback for handling directives is the same as the name of the directive, the `func` key allows us to point to a different subroutine instead. Here we will be reusing `AuthDigestProvider()` to process both directives. How will we know which directive is actually being parsed? The `cmd_data` slot will contain `digest` when processing `AuthDigestProvider` and `basic` when processing `AuthBasicProvider`.

At this point, we have defined what our directives will look like and how they will interact with Apache in `httpd.conf`. What we have not shown is the logic that sits behind our directives. As we mentioned, both of our directives will be calling the Perl subroutine `AuthDigestProvider`, defined in `AuthenHook.pm`.

```
sub AuthDigestProvider {
  my ($cfg, $parms, $args) = @_;

  my @providers = split ' ', $args;

  foreach my $provider (@providers) {

    # if the provider looks like a Perl handler...
    if ($provider =~ m/::/) {
      # save the config for later
      push @{$cfg->{$parms->info}}, $provider;

      # and register the handler as an authentication provider
      register_provider($provider);
    }
  }

  # pass the directive back to Apache "unprocessed"
  return Apache::DECLINE_CMD;
}
```

The first argument passed to our directive handler callback, `$cfg`, represents the configuration object for our module, which we can populate with whatever data we choose and access again at request time. The second argument is an Apache::CmdParms object, which we will use to dig out the string we specified in the `cmd_data` slot of our configuration hash using the `info()` method.

While the first two arguments are standard and will be there for any directive handler you write, the third argument can vary somewhat. Because we specified `RAW_ARGS` as our `args_how` setting in the configuration hash, `$args` contains everything on the `httpd.conf` line following our directive. The standard `Auth*Provider` directives we are overriding can take more than one argument, so we split on whitespace and break apart the configuration into an array of providers, each of which we then process separately.

Each provider is examined using a cursory check to see whether the specified provider is a Perl provider. If the provider meets our criteria, we call the `register_provider()` function defined in `AuthenHook.xs` and keep track of the provider by storing it in our `$cfg` configuration object.

The final part of our callback brings the entire process together. The constant `DECLINE_CMD` has special meaning to Apache. Just as you might return `DECLINED` from a `PerlTransHandler` to trick Apache into thinking no translation took place, returning `DECLINE_CMD` from a directive handler tricks Apache into thinking that the directive was unprocessed. So, after our `AuthDigestProvider()` subroutine runs, Apache will continue along until it finds `mod_auth_digest`, which will then process the directive as though we were never there.

The one final piece of `AuthenHook.pm` that we need to discuss is directive merging. In order to deal properly with situations when directives meet, such as when `AuthBasicProvider` is specified in both an `.htaccess` file as well as the `<Location>` that governs the URI, we need to define `DIR_CREATE()` and `DIR_MERGE()` subroutines.

`DIR_CREATE()` is called at various times in the configuration process, including when `<Location>` and related directives are parsed at configuration time, as well as whenever an `.htaccess` file enters the request cycle. This is where we create the `$cfg` object our callback uses to store configuration data. While it is not required, `DIR_CREATE()` is a good place to initialize fields in the object as well, which prevents accidentally dereferencing nonexistent references.

```
sub DIR_CREATE {
  return bless { digest => [],
                 basic  => [], }, shift;
}
```

`DIR_MERGE`, generally called at request time, defines how we handle places where directives collide. The following code is standard for allowing the current configuration (`%$base`) to inherit only missing parameters from higher configurations (`%$add`), which is the behavior you are most likely to want.

```
sub DIR_MERGE {
  my ($base, $add) = @_;

  my %new = (%$add, %$base);

  return bless \%new, ref($base);
}

1;
```

Thus ends `AuthenHook.pm`.

The final result is pretty amazing. By secretly intercepting the `AuthDigestProvider` directive before `mod_auth_digest` has the chance to process it, we have provided an interface that makes the presence of Apache::AuthenHook all but undetectable. To enable the new provider mechanism for mod_perl developers, all that is required is to load Apache::AuthenHook using the new `PerlLoadModule` directive

```
PerlLoadModule Apache::AuthenHook
```

and their Perl providers will be magically inserted into the authentication phase at the appropriate time.

Taking a Step Back

Let's recap what we have accomplished so far. `AuthenHook.pm` redefines `AuthDigestProvider` and `AuthBasicProvider` so that any Perl providers listed in the configuration are automagically registered and inserted into the authentication process. At request time, one of the default Apache authentication handlers will call on the configured providers to supply server-side credentials. All registered Perl providers really point to the callbacks in `AuthenHook.xs` which have the arduous task of proxying the request for server-side credentials to the proper Perl provider. All in all, Apache::AuthenHook covers lots of ground, even if the gory details of what happens over in XS land have been left out.

As we mentioned earlier, Apache::AuthenHook not only the ability to write authentication providers in Perl, but it also follows the Apache API very closely. While diving deep into the XS code that Apache::AuthenHook uses to implement the `check_password` and `get_realm_hash` callbacks is far beyond the scope of this article, you may find it interesting that the callback signature for `check_password`

```
static authn_status check_password(request_rec *r, const char *user,
                                   const char *password)
{
  ...
}
```

is practically identical to what Apache::AuthenHook passes on to Perl providers supporting the Basic authentication scheme.

```
sub handler {
  my ($r, $user, $password) = @_;

  ...
}
```

If you recall, we started investigating the Apache 2.1 provider mechanism as a way to combine the security of the Digest authentication scheme with the strength of Perl. The signature for the Digest authentication callback, `get_realm_hash`, is only slightly different than `check_password`.

```
static authn_status get_realm_hash(request_rec *r, const char *user,
                                   const char *realm, char **rethash)
{
  ...
}
```

How does this translate into a Perl API? It is surprisingly simple. As it turns out, the name `check_password` is significant—for Basic authentication, the provider is expected to take steps to see if the incoming username and password match the username and password stored on the server back-end. For Digest authentication, as the name `get_realm_hash` might suggest, all a provider is responsible for is retrieving the hash for a user at a given realm. `mod_auth_digest` does all the heavy lifting.

Digest Authentication for the People

While we didn't take the time to explain how Basic authentication over HTTP actually works, briefly explaining Digest authentication is probably worth the time, if only to allow you to appreciate the elegance of the new provider mechanism.

When a request comes in for a resource protected by Digest authentication, the server begins the process by returning a `WWW-Authenticate` header that contains the authentication scheme, realm, a server generated nonce, and various other bits of information. A fully RFC-compliant `WWW-Authenticate` header might look like the following.

```
WWW-Authenticate: Digest realm="realm1",
nonce="Q9equ9C+AwA=195acc80cf91ce99828b8437707cafce78b11621",
algorithm=MD5, qop="auth"
```

On the client side, the username and password are entered by the end user based on the authentication realm sent from the server. Unlike Basic authentication, in which the client transmits the user's password practically in the clear, Digest authentication never exposes the password over the wire. Instead, both the client and server handle the user's credentials with care. For the client, this means rolling up the user credentials, along with other parts of the request such as the server-generated nonce and request URI, into a single MD5 hash, which is then sent back to the server via the `Authorization` header.

```
Authorization: Digest username="user1", realm="realm1",
qop="auth", algorithm="MD5", uri="/index.html",
nonce="Q9equ9C+AwA=195acc80cf91ce99828b8437707cafce78b11621",
nc=00000001, cnonce="3e4b161902b931710ae04262c31d9307",
response="49fac556a5b13f35a4c5f05c97723b32"
```

The server, of course, needs to have its own copy of the user credentials around for comparison. 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 hash generated by the server does not match the one sent by the client in the `Authorization` header, the difference can be attributed to the one piece of information not mutually agreed upon through the HTTP request: the password.

As you can see from the headers involved, there is quite a lot of information to process and interpret with the Digest authentication scheme. However, if you recall, one of the benefits of the new provider mechanism is that `mod_auth_digest` takes care of all the intimate details of the scheme internally, relieving you from the burden of understanding it at all.

All a Digest provider is required to do is match the incoming user and realm to a suitable digest, stored in the medium of its choosing, and return it. With the hash in hand, `mod_auth_digest` will do all the subsequent manipulations and decide whether the hash the provider supplied is indeed sufficient to allow the user to continue on its journey to the resource it is after.

With that background behind us, we can proceed with a sample Perl Digest provider.

```
package My::DigestProvider;

use Apache::Log;

use Apache::Const -compile => qw(OK DECLINED HTTP_UNAUTHORIZED);

use strict;

sub handler {
  my ($r, $user, $realm, $hash) = @_;

  # user1 at realm1 is found - pass to mod_auth_digest
  if ($user eq 'user1' && $realm eq 'realm1') {
    $$hash = 'eee52b97527306e9e8c4613b7fa800eb';
    return Apache::OK;
  }

  # user2 is denied outright
  if ($user eq 'user2' && $realm eq 'realm1') {
    return Apache::HTTP_UNAUTHORIZED;
  }

  # all others are passed along to the next provider
  return Apache::DECLINED;
}

1;
```

Note the only slight difference between the interface for Digest authentication as compared to Basic authentication. Because the authentication realm is an essential part of the Digest scheme, it is passed to our `handler()` subroutine in addition to the request record, `$r`, and username we received with the Basic scheme.

Knowing the username and authentication realm, our provider can choose whatever method it desires to retrieve the MD5 hash associated with the user. Returning the hash for comparison by `mod_auth_digest` is simply a matter of populating the scalar referenced by `$hash` and returning `OK`. While using references in this way may feel a bit strange, it follows the same pattern as the official Apache C API, so I guess that makes it ok.

If the user cannot be found, the provider can choose to return `HTTP_UNAUTHORIZED` and deny access to the user, or return `DECLINED` to pass authority for the user to the next provider. Remember, unlike with the Perl handlers for all the other phases of the request, you can intermix Perl providers with C providers, sandwiching the default file provider with Perl providers of your own choosing.

The one question that remains is how to generate a suitable MD5 digest to pass back to `mod_auth_digest`. For the default file provider, the return digest is typically generated using the `htdigest` binary that comes with the Apache installation. However, a Perl one-liner that can be used to generate a suitable MD5 digest for Perl providers would look similar to the following.

```
$ perl -MDigest::MD5 -e'print Digest::MD5::md5_hex("user:realm:password")'
```

That is all there is to it. No hash checking, no header manipulations, no back flips or somersaults. Simply dig out the user credentials and pass them along. At last, Digest authentication for the (Perl) people.

Don't Forget the Tests!

Of course, no module would be complete without a test suite, and the Apache-Test framework introduced last time gives us all the tools we need to write a complete set of tests.

For the most part, the tests for Apache::AuthenHook are not that different from those presented before. LWP supports Digest authentication natively, so all our test scripts really need to do is make a request to a protected URI and let LWP do all the work. Here is a snippet from one of the tests.

```
plan tests => 10, (have_lwp &&
                   have_module('mod_auth_digest'));

my $url = '/digest/index.html';

$response = GET $url, username => 'user1', password => 'digest1';
ok $response->code == 200;
```

When we plan the tests, we first check for the existence of `mod_auth_digest`—both `mod_auth_basic` and `mod_auth_digest` can be enabled or disabled for any given installation, so we need to check for them where appropriate. Passing the username and password credentials is pretty straightforward, using the `username` and `password` keys after the URL when formatting the request.

Actually, while the `username` and `password` keys have special meaning, you can use the same technique to send any arbitrary headers in the request.

```
# fetch the Last-Modified header from one response...
my $last_modified = $response->header('Last-Modified');

# and use it in the next request
$response = GET $uri, 'If-Modified-Since' => $last_modified;
ok ($response->code == HTTP_NOT_MODIFIED);
```

That's something to note just in case you need that functionality sometime later in your testing life.

One final note about our tests will apply to anyone writing a mod_perl XS extension. Instead of using `extra.conf.in` to configure Apache, we used `extra.last.conf.in`. The difference between the two is that `extra.last.conf.in` is guaranteed to be loaded the last in the configuration order—if our `PerlLoadModule` directive is processed before mod_perl gets the chance to add the proper `blib` entries, nothing will work, so ensuring our configuration is loaded after everything else is in place is important.

Whew

mod_perl is truly exciting. With surprisingly little work, we have managed to open an entire new world within Apache 2.1 to the Perl masses. I know of no other blend of technologies that allow for such remarkable flexibility beyond what each individually brings to the table. Hopefully, this article has not only introduced you to new Apache and authentication concepts, but has also brought to light ways in which you can leverage mod_perl that you never thought of before.

More Information

I apologize if this article is a little on the heavy side, teasing you with only cursory introductions to cool concepts while leaving out the finer details. So, if you want to explore these concepts in more detail, I leave you with the following required reading.

A nice overall introduction to the new provider mechanism can be found in Safer Apache Driving with AAA. The mechanics of Basic authentication can be found in lots of places, but decent explanations of Digest authentication are harder to find. Both are covered to some level of detail in Chapter 13 of the *mod_perl Developer's Cookbook*, which is freely available online. Recipe 13.8 in particular includes the code that became the splinter in my mind and eventually this article.

A more detailed explanation of directive handlers in mod_perl 2.0 can be found in the mod_perl 2.0 documentation. Although covering only mod_perl 1.0 directive handlers, whose implementation is very different, Chapter 8 in *Writing Apache Modules with Perl and C* and Recipes 7.8 through 7.11 in the *mod_perl Developer's Cookbook* provide excellent explanations of concepts that are universal to both platforms, and are essential reading if you plan on using directive handlers yourself. If you are curious about the intricate details of directive merging, Chapter 21 in Apache: the Definitive Guide presents probably the most comprehensive explanation available.

Finally, if you are interested in the gory details of the XS that really drives Apache::AuthenHook, there is no better single point of reference than *Extending and Embedding Perl*, which was my best friend while writing this module and absolutely deserves a place on your bookshelf.

Thanks

Many thanks to Stas Bekman and Philippe Chiasson for their feedback and review of the several patches to mod_perl core that were required for the code in this article, as well as to Jörg Walter, who was kind enough to take the time to review this article and give valuable feedback.