

Hardening HTAccess, Part Two

Robert Hansen 2001-07-25

Hardening HTAccess, Part Two

by Robert Hansen

last updated July 25, 2001

This article is the second in a three-part series that will provide a way to harden htaccess to make it more stable and lessen the chances of successful brute force attacks. The [first installment](#) offered a brief overview of htaccess, along with a look at a couple of hacking tools and methodologies to which htaccess is particularly susceptible. We particularly covered ways in which wwwhack can be used to infiltrate htaccess. In this article, we will explore a couple of ways of foiling [wwwhack](#), namely headers masking and content masking.

Headers Masking

Let's review the first method that can be used with wwwhack to crack user authentication. In many wwwhack tests, I have used a CGI script to return false headers, but have likewise run into many obstacles. The idea being to try to match the return headers regardless of the real answer. There are two ways of doing this. The first way was to return a 401 header regardless of whether the user is truly authenticated or not. This can be accomplished by printing out "Status: 401 Authorization Required\nContent-Type: text/html\n\n" before printing out the rest of the script.

(Note: '\n' is a new line character, but this is not really printed to the socket as seen, as there is a lot of behind-the-scenes processing done by Apache and it usually ends up being a lot more outputted text. If you want to see what is happening, telnet to port 80 of the web server, type "GET /{script name} HTTP/1.0", and hit return twice. As a matter of style and in order to protect yourself, when dealing with authentication, I suggest doing all the processing in your CGI prior to printing. In almost all other cases, this should be reversed to speed up browser rendering while the script is processing; however, this is a matter of semantics and is outside the realm of this discussion.)

This worked fine for Internet Explorer but not for Netscape because the latter stops sending username/password combinations. Internet Explorer works incorrectly in that it keeps sending the username/password information regardless of the header, but this has the side-effect of making this hack work properly. It has been suggested to me that browser detection might be a good way around this, but it wouldn't work in the long-term, because it wouldn't take much work to change the browser-type spoofed by wwwhack to say it was Netscape. So scrap that option for now.

The second option to defeat wwwhack was to return 200 OK in all cases (this can be accomplished in the same way as above: by printing "Status: 200 OK\nContent-Type: text/html\n\n" at the beginning of the script.) This has some interesting side effects. The first being that the username/password box doesn't pop up. The second being that sending username/passwords via Basic Auth becomes impossible. So it has the

effect of making it totally impossible to enter a site without modifying the URI (EG: `http://username:password@www.site.com/private/`) string with a normal browser. That's no good either.

So I took another approach. In reality, all that needs to happen is that the web-server agrees that you have the proper credentials to enter a site. If Apache could be made to believe that you are an authorized user, you can enter the site. What is authorization? Previously we thought that authorization was when a user typed his username/password combination into the browser's pop-up and that got translated into a `Auth-Type Basic` header that then was used by the browser for every html page, graphic and piece of multi-media in the directory hierarchy. What if we thought about authentication in the form of a token? This might solve a number of problems.

Still with me? It gets worse. If the `index.cgi` page returns a cookie that can be used as an authentication token for the rest of the `htaccess`-protected directory, we could subvert the problem of needing to send the password over and over again. The browser will pass the cookie for each request to the web server. We can have a program that runs interference between the authentication that takes the place of normal `htaccess` authentication. To clarify, each hit to the `index.cgi` program would return a new cookie. That cookie would be an authentication token to be used throughout the site. If the user doesn't have this cookie, but instead posts a correct username/password combination to the `check.cgi` script, this will allow the user to view the password-protected directory and get the correct credentials.

One way to do this is to use the [Apache module `mod_auth_external`](#) which will allow you to run a separate script in place of Apache's `.htpasswd` checking algorithm. There are other ways to do this, which I will mention later, but this is the easiest by far. According to the website, `mod_auth_external` does not work with the Windows version of Apache. There are alternatives, but this will not be discussed at length.

So you can have the 401 error program (defined below) give a cookie to the browser (the reason for this will be described later), and then when the sub-frame section (heretofore called the `mainframe.html` portion) of the 401 error page is called it will print a form submission box for username and passwords. So the `mainframe.html` portion of the frame-set will be our login page.

Since the user has not yet been authenticated, if the user attempts to go to `www.yoururl.com/private/index`.

cgi they will see `www.yoururl.com/401.cgi` but the URI string will still say `www.yoururl.com/private/index.cgi` regardless. This can be used to our advantage. We'll come back to that.

The path to `401.cgi` is defined in the `.htaccess` document. `mainframe.html` has a form submission box (username and password fields) that it will post to `check.cgi`. `check.cgi` will be located outside the password protected directory. `check.cgi` will either authenticate the user in which case a local database will get updated with the cookie they currently have as a temporarily valid cookie to be replaced by a more persistent cookie placed by `/private/index.cgi`. If they aren't authenticated `check.cgi` logs the attempt and posts them to `/private/index.cgi` which will send them to `/401.cgi`. The reason being, if `check.cgi` were to post them to `401.cgi` directly, the brute force program could detect the difference in URLs. Tricky, I know. This is how `401.cgi` is defined in the `.htaccess` file:

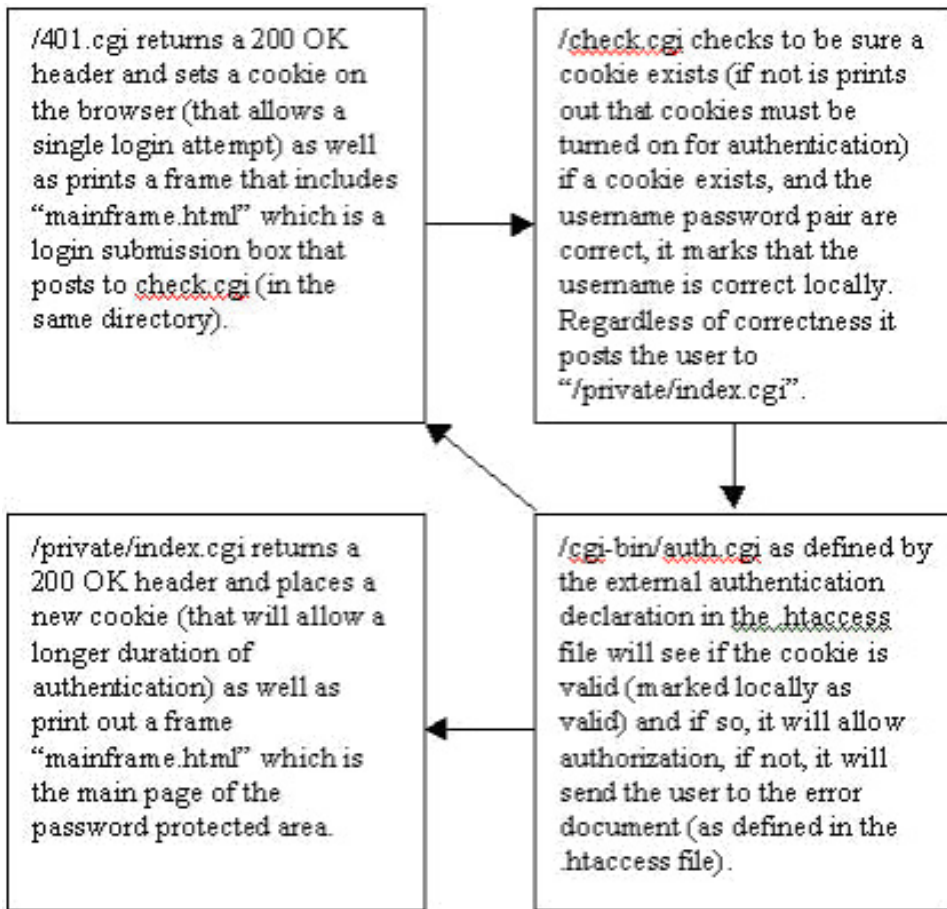
```
ErrorDocument 401 /401.cgi
```

The environmental information (the POST information, cookies, browser types, etc...) will get passed to the `401.cgi` script which can be used in the generation of a new cookie to be given to the user. The idea being that the cookie needs to be unique. This cookie could be thought of as an "attempt cookie" valid for only one username/password attempt. The "attempt cookie" token must be based somewhat on time so there is no risk of redundant cookies. If they have this "attempt cookie" and a valid username/password combination they will also be allowed to enter.

If you have made it this far in the paper, you have made it through the worst of it.

When they enter the site, the index program (`index.cgi`) will give them a cookie based on some credentials, such as `/username/browser-type/time` etc., in place of the one they had received before.

Whew! I think I need a diagram here.



(Note: I am not going to talk about the algorithm used because that will be based on the user's particular needs and the level of security that will be desired. If you really must know more about this particular topic, you can either [contact me](#) or read books like [Network Security: Private Communication in a Public World](#) by Kaufman, Perlman and Speciner.)

The reason you want a cookie to be placed with the browser for the 401 script is a complex one. You want to mirror what happens when you get a valid password response with what you get in a false one. Imagine if you got a cookie with one and not with the other? Obviously that is something that can easily be searched for, in the same way headers and text can be searched for. So the cookies have to match in format, size and name, but one has to be wrong, and the other right, (or logged as invalid or valid respectively) according to your own encryption algorithm or token schema.

(Note: there is the issue about the programming language used. You can use whatever best suits you but of course there will be performance issues with some languages over others. Also worthy of mention is the fact that the `getenv()` function (used for CGI) in C is insecure as it doesn't have bounds checking and is therefore a candidate for buffer overflows. This was proven in the CrazyWWWBoard CGI exploit, as well as the lynx local `getenv()` exploit among others. However, that is a religious issue that is better left for a forum other than this paper.)

Another positive that can be derived from this is that you are no longer passing sensitive information over and over again. The username and password combination is only sent once, and the cookie should become

invalid if not used within a certain amount of time. Your authentication program can add a time-stamp of the last access via that cookie to a database, and then can time out any invalid sessions. In this way, htaccess could actually be as secure as many homegrown Apache authentication modules.

(As a side-note: to the best of my knowledge changing the headers to include 200 OK and 401 Unauthorized both will fool RFP's whisker into believing all requests are unauthorized but will not pop up the htaccess windows under Netscape and Internet Explorer. I have not brought this particular point into my paper because htaccess hardening does work to prevent whisker Basic-Auth brute force attacks.)

With this method of using wwwhack satisfactorily solved, we no longer have the problem of runaway processes or too many zombied processes because there is no more htaccess pop-up from the browser, so that the headers checking method cannot be used. Now wwwhack will have to be set to view all the content returned from the page, which in turn will allow our scripts and Apache to clean up nicely after themselves. Now we just have to deal with the issue of too many resources being used when wwwhack is running its course checking for strings. This brings us to the next section.

Content Masking

As you will recall the second method that wwwhack uses to see if it has found a valid password, is looking at the content. Wwwhack doesn't have any natural language processing engine, or anything equally smart. It simply checks the text and sees if it matches. What we need is to make certain that the text in the 401 document matches what is returned by index.cgi script, exactly. Exactly? How can it be exactly and still look different to the user? The answer is frames. Frames seem kind of clumsy - then again, so do cookies, but let's look at the facts.

Almost every modern browser uses frames and uses cookies - even the terminal-based [Unix program Links](#) can view tables and use cookies. We lose some small measure of compatibility when we use these sorts of technology. Not to downplay the importance of compatibility but we gain something that could not otherwise be gained with such a low cost to develop. Mind you, this document is most certainly not meant for the high-level developer who has months of time to develop a good authentication system, and this is certainly not better than combining one-time-pad type certificates with home-brew Apache modules, etc. In a real-world environment people want a quick, dirty solution that requires next to no development for a high gain in profitability. This is the issue I am addressing.

So, by using frames we can have the exact same text in both returned documents. The mainframe.html frame (and only frame unless you have some reason to have multiple frames) will be a link to the login page in the 401.cgi script, and a link to the main private page in the index.cgi script. This gets tricky. If the password is wrong, the external authentication program will simply repost the user back to 401.cgi (via the redirect in the .htaccess defined as the 401 ErrorDocument). By the way, it is important that you configure Apache to use .cgi as a index file for this to work. The following is an example of how:

```
DirectoryIndex index.html index.htm index.cgi
```

(Note: If you don't want to put CGI scripts all over your site, you can use Apache aliases to point a supposed CGI script in a password protected directory to your global cgi-bin directory, if you are paranoid about such things. This is the Apache equivalent to symbolic links, although slightly more powerful. You can do things like point a directory /asdf/ to /cgi-bin/asdf.cgi so you could have stuff like have a link to /asdf/blah.js which is actually a link to asdf.cgi and then have blah.js read in by the PATH_INFO environmental variable and print out java script on the fly by a CGI program. I've written a reseller program where "blah" was replaced by the reseller ID (and some other information about the types of banners requested). I was able to dynamically create the banners (dynamically like anything, tables, multi-media, pop-up windows, etc...) with just a single "<SCRIPT SRC>" tag based off the user's preferences. Pretty slick, huh?)

The authentication function defined by mod_auth_external also needs to check to see if the cookie is there and valid. If the credential isn't there, a message about having cookies as a requirement should be displayed. If it is there, but it's invalid, the password login page should pop-up. This means that the cracker can't guess the password without getting the exact same thing returned. A recursive function would have to be written to see the mainframe.html portion of the frame/script that is returned by index.cgi and/or 401.cgi. Not that this is impossible, but there are ways around it, like using remote java-script so the cracking program must then follow all links off of mainframe.html to be certain these sub-frames do not contain the keywords the cracking program is looking for to be certain that access has not been denied.

Another important thing to mention is to continue sending new cookies via the 401 script and with index.cgi both. Otherwise the cracker would be able to detect that the cookie has not changed. This might feel like a little bit of overhead, and you can decide if it is worth it to you or not. Also, the 401.cgi program must be viewed (must pass a logical cookie to the browser) before attempting to login via posting the username/password pair to the protected index CGI script to insure that the browser has actually seen the login page prior to posting. The idea here is that we are making the most amount of overhead possible for the attacker in an attempt to slow them down.

Here is an example of what 401.cgi could look like in PERL (I will be using PERL throughtout this paper because of it's quick time to develop and because it is designed to work well with CGI). The reason I use REQUEST_URI instead of QUERY_STRING is because QUERY_STRING is not passed via 401 unauthorized. This does not include the cookie placing portion of the code, as that procedure is based largley on the implementation. index.cgi will look very similar. I am leaving the inner workings of this program in a PERL module called AuthConfig.pm left as a test to the reader to write.

```
use AuthConfig; #I am abstracting this from a module for ease of reading.  
  
my $auth = new AuthConfig;
```

```
$sessionid = $auth->set_session_id(); #this will be created dynamically

#REQUEST_URI is a final translation done by Apache of the URI so
#character substitution attacks and other things described in whisker's
#documentation have no relevance to this pattern match

if ($ENV{'REQUEST_URI'} =~ /\?mainframe.html$/){

    print "Status: 200 OK\n";

    print "Content-Type: text/html\n\n";

    $auth->print_mainframe_html();

} else {

    print "Status: 200 OK\n";

    print "Set-Cookie: sessionid=$sessionid; path=/\n"; #you don't need CGI.pm to
write cookies

    print "Content-Type: text/html\n\n";

    $auth->print_frameset(); #this html is described above

}
```

Also, remember that because we are dealing with frames all links off of both version of mainframe.html (in the non-password protected as well as the password protected areas) will need to have the target pointed to "_top" as to refresh the entire frame, and not just the mainframe.html documents.

Okay, so two issues down out of two, we are out of the woods, right? Wrong. Unfortunately, we are out of time and space. We will discuss more issues involved with hardening htaccess in the next, and last, installment of this series.?

To read **Hardening HTAccess, Part Three**, click [here](#).

[Privacy Statement](#)

Copyright 2006, SecurityFocus