

Hardening HTAccess, Part One

Robert Hansen 2001-07-11

Hardening HTAccess, Part One

by *Robert Hansen*

last updated July 11, 2001

Introduction

Htaccess can be used to manage multiple usernames/passwords, thereby enhancing information protection on the web server by controlling access through HTTP protocols. When used in conjunction with a browser encryption method such as SSL, it is possible to make htaccess authentication a robust method of protecting directories. However, out of the box, htaccess is prone to several problems, namely: packet-sniffing, IP hijacking, replay attacks, and brute force. Cryptography, (SSL and one-time pads) can solve all but one of these problems - brute forcing.

Brute forcing takes a number of forms, and is a well-known and well-used attack against htaccess. Brute force is usually a minimal knowledge attack, requiring only the URL for the password-protected directory to work. In their most malevolent form, brute force attacks simply check the headers returned by the server. If the program sees that its request was favorable (the server returned a 200 OK response), it will mark the password as being valid. This can wreak havoc on a server. It can even cause denial of service when the brute force program disconnects after viewing the headers (as the server is not allowed to print out the rest of the content and the daemon cannot kill its children efficiently.)

This article is the first in a three-part series that will provide a way to harden htaccess (in coordination with a SSL and a one-time pad token type authentication mechanism) to make it more stable and lessen the chances of successful brute force attacks. This installment will offer a brief overview of htaccess, particularly why it is prone to attacks by brute force, and a look at a couple of hacking tools and methodologies to which htaccess is susceptible.

The Problem

There are literally dozens of on-line documents on securing htaccess (hyper-text access) scripts and directories. There are equally as many papers that have a large portion of the topic glossed over or missing on the topic. Most of them go along the lines of:

- make sure unauthorized users cannot read the .htaccess script by using the httpd.conf or httpds.conf file to secure any files beginning with '.ht';
- make sure all '.ht' files are readable by the server;
- make sure the right deny/allow order is set;
- etc., etc.

By the way, in case you didn't know how to do this here is a config file snippet:

```
Order allow, deny
Deny from all
```

Note: The reason .htaccess starts with a '.' is because in Unix it is a tradition that files starting with a '.' are "hidden" files or configuration files that aren't supposed to be edited by anyone who doesn't know better. Really this doesn't mean anything other than that users have to type a switch when doing a directory listing to view these files. It is a convention worth noting, however, as even its name implies that it shouldn't be viewed.

Users will also need to have this line in the httpd/s.conf file to tell Apache to look at the .htaccess file for access control information:

```
AccessFileName .htaccess
```

That stuff is all fine and dandy, but what are the real life concerns of htaccess? What are its real strengths and weaknesses?

There are numerous ways to secure data, but the only way to secure large quantities of data and data types (this is important) is to either set up scripts to act as a proxy script - to filter who can and cannot see the data - or to use something more global like a password protected directory. The standard out-of-the box method to password protect directories is to use an .htaccess script. In reality, this is exactly the same thing as setting up a script to monitor who can and cannot view data. But let's suppose, for the sake of this paper, that it is different.

The typical system administrator or web-application engineer can look at each of these options and, depending on his or her skill level and time constraints, will more often than not decide on the easier out-of-the-box solution provided by htaccess scripts. These are well supported and

well documented by both [Apache](#) and [Zeus](#) web servers. (Due to its popularity and the fact that it is open source, I'll be using Apache in this series to explain htaccess.)

The problem is not that htaccess is unsupported, but that it is misunderstood. The benefits of htaccess are that it is easily maintained, easily understood (at least on how to write .htaccess scripts), and the two major browsers (IE and Netscape) have very predictable responses in success and failure of password authentication. Well, this is true in almost all cases, but we are getting ahead of ourselves.

A Hypothetical Attack Scenario

How might a brute force attack be perpetrated, and how might it appear to someone charged with monitoring the system under attack? Let's say take a hypothetical case of a system administrator who needs to protect some subscription-based content on his web site (our hypothetical administrator is a male.) Some other programmer somewhere has developed the billing, and can add and delete passwords from a database reliably. The administrator chooses an htaccess script to protect the subscription content because everyone seems to support it and it is the easiest choice.

Everything seems to be going fine, the site is getting subscriptions to its content, and it doesn't let anyone in who doesn't have a password. All of a sudden at 6PM EST, the admin is being paged and called on the phone because something is happening to the server. The admin gets on the lap-top and connects to the web-server. He tries to [SSH](#) in and is surprised to find that the web server is up. The admin then checks to make sure Apache is still running, and again is surprised to find that it is. After checking around, he finds that there are a few hundred connections, zombied processes; the load is pretty high, and he can't figure out what it is.

Scratching his head, the admin shuts off the web server, crosses his fingers, and happily finds that the load goes back to normal after a minute. He restarts Apache, and it seems okay. However, a few minutes later the same thing happens. The administrator is dumbfounded, until he realize that almost all the connections are coming from the same IP address - if he's lucky.

Our hypothetical admin realizes that someone is trying to break through the htaccess script and so he IP-chains the cracker off the site. He waits a few minutes. Everything seems fine - except, of course, for the fact that without some digging he won't know which passwords have been cracked. Also, who knows how much downtime he might have incurred, not to mention

lost revenue, a screaming boss, and who knows what else. This is really a best-case scenario believe it or not, and I will go into why, but let's take this as an example of a fairly typical attack.

What is wwwhack?

What type of tool might have been used to perpetrate this brute force? One such tool is wwwhack. Wwwwhack is a tool that can be sometimes downloaded from <http://www.wwwhack.com>; however, it is often more reliable to search for wwwhack as the site is often down or out of date. Wwwwhack takes a dictionary file and tries to brute force Basic Auth type authentication over http. That is, it sends a bunch of username/password pairs at the web server to test the . htpassword file (or password database or however you have this set up) for correct username/password pairs.

Note: There are many Auth-Type Basic cracking utilities out there, but wwwhack is really one of the best one out there, and works very well for this discussion because of the two tests it can run against the data returned. Another program worthy of mention is [Whisker](#) written by Rain Forest Puppy to find common CGI vulnerabilities. Wwwwhack will be used as an example in this discussion because it is not particularly complex.

Cracking Methodology

The first mode wwwhack can run in (and the more damaging of the two) is the mode which checks headers to see if the web server returns 200 OK (the password is correct) or 401 Authorization Required (the password is not passed correctly or it is incorrect). This method can wreak havoc on a web server. Let's say for instance that the index file is called index.cgi and is in fact a CGI script instead of an HTML file, and is thus interpreted.

This is a fairly common example, and what will end up happening is that since wwwhack doesn't clean up nicely, and doesn't really care about the data returned to it, it will close the connection prior to Apache completing the transaction and finishing the output of the program. This can cause zombie processes (where the child can no longer speak to the parent, and has no socket to talk to and just basically hangs, waiting for the parent to kill it). This eventually will cause a spike in load. This has been demonstrated with medium-sized web servers - dual 400Mhz processors with 1 gig of RAM with less than 60k hits a day - to completely deny service to the web-server (in the sense that reliable service is non-existent).

Some people say this can be solved by some tricky Apache config stuff, to get Apache to clean up its processes much quicker. Here are a few examples of things you could change (these are the defaults the author found in his particular httpd.conf file):

```
Timeout 300
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 15
MinSpareServers 5
MaxSpareServers 10
MaxClients 150
MaxRequestsPerChild 0
```

Readers should look at these directives, as there is useful information in the httpd/s.conf file about these parameters; however, limiting policy using these variables can hurt legitimate traffic, as well as crackers. These are temporary fixes, in a lot of ways, but could certainly save readers a lot of time and trouble while they are developing a more robust solution. These directives have comments in the httpd/s.conf file about the purpose of each variable.

There are several problems with changing these items in the long run. There may be some traffic that legitimately had trouble connecting, (for latency reasons, or slow connection speed, etc...) that might otherwise get cut-off with such a stringent policy. Also, configuration changes do not generally solve the problem under high-load, they just prolong the life of the web-server a little while longer (not to say that this is a bad thing). We need to find a reasonable solution to this problem that doesn't hurt legitimate users and will not allow this kind of activity as a matter of policy.

The second mode in wwwhack is a server-friendly mode, as it actually allows the entire connection to complete. This means that it checks to see if it finds a string such as "Access Denied" or "Enter your password again" in the page that is returned. If it does find this string it continues to try usernames and passwords until it does not find the string, at which point it marks that username and password. It then may or may not continue to find more username/password pairs depending on the configuration. The important part is that it actually downloads all the content before the socket closes (in my experience). This will not cause zombie processes. It may fill up the amount of connections an admin allows at a given time and it may allow the cracker to find a username/password pair that works, but it will not cause tremendous spikes in load, IPC (Inter-Processes Communication) locking problems, or database problems.

This is a good thing. Needless to say, this string hunting method is less efficient, and is less likely to be used if the headers checking method is an alternative.

In the [Next Installment...](#)

As stated previously, this article is the first in a three-part series. This installment has offered a brief overview of htaccess, particularly why htaccess is prone to attacks by brute force, and a couple of methodologies that can be used to crack authentication in htaccess. The next article in the series will examine headers masking and content masking.

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

[Privacy Statement](#)

Copyright 2006, SecurityFocus