

## Hardening HTAccess, Part Three

*Robert Hansen* 2001-08-06

### Hardening HTAccess, Part Three

by *Robert Hansen*

last updated August 6, 2001

---

This is the third and final installment in a series devoted to hardening 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 the [second article](#), we explored a couple of ways of foiling wwwhack, namely headers masking and content masking. This installment will look at a few more issues involved with hardening htaccess.

#### More Issues?

The entrance point of an htaccess protected directory does not have to be the first page. Intruders could, for instance go to an image. If they knew where the image was, they could go straight to that file. They could spoof the 401 header to say 200 but the content type would not make sense (returning text/html when you asked for a JPG doesn't make a whole lot of sense does it?). Therefore users probably will need to include something that checks the extension to make certain the return content type matches the type of file that is being requested.

An important note I haven't yet stressed is that it is imperative that the authentication program checks to make certain the cookie is legitimate, or, if there is are POST arguments, the arguments (username/password) must be legitimate when we are talking about check.cgi.

Here we run into a real problem. If the user has a legitimate username, they can see any file in the directory. That might seem obvious but what if wwwhack was set to test usernames and passwords inside a directory like /secure/images/secureimages.jpg. That file does not include some frame-set that is returned by the 401 error script. If the person has a legitimate account, they can see any file in the directory, and therefore can check for strings (or the lack of strings) in the exact same way that it was able to before.

This means that we would have to spoof the content for every single page in order to legitimately make it impossible for wwwhack to tell the difference between a 401 error and a

200 okay. This would be akin to having no security at all, as you would be returning the content for every request in your 401 error. Confusing, I know, but because of this reason, we have to ignore the fact that users who do have access to the protected area can tell wwwhack to check other files than the index file.

For this reason, it would seem that htaccess hardening has a flaw, wouldn't it? To properly exploit this flaw, however, the user does have to have access to the protected directory (or they would have to try to detect 404 errors, which we could take care of in the same way we took care of 401 errors), and (here's the first catch) we would have to let the cracker try as many guesses as they choose. The second catch is that we would have to let them deep-link without having a cleared token that we have previously (this is important) agreed is valid. Reread that last sentence.

(Note: It is a good idea to do the same thing with 404 errors, and this will require another script (called something like /cgi-bin/404.cgi) that returns 200 okay headers and then tells the user that they have reached a document that is not found. You don't want to try to make them log-in again, as that might be confusing, and it might be your fault a link is broken for legitimate users. The 404 errors should not have the frame that 401 does, because it will need to be informative to the user who found a broken link, and 401 errors will take precedence anyway.)

As an aside of sorts, I had also thought that the possibility of encrypting all non HTML/JavaScript, etc. with the ASCII hex equivalent of the original character. Then take that same code and all possible variants and place that same text in the mainframe.html version of the index.cgi script to confuse any pattern matching, and hopefully any natural language processors as well. That is, a space, " " would be turned into "&20;" an "a" would turn into "&41;" and so on. (Look on this [ASCII resource web page](#) for a full list of characters). Later, I realized that this is overkill, and would have negligible gain in the long run, especially for users who have access to the protected area of the web site. There must be a better choice, right?

If you check the amount of times an IP address has been accessing the web site (and they have a cookie that is still valid, which we will talk about in a second) over a given time - say five minutes - you can deny them access via the computer or the web-server. Okay, so here, the administrator should include something in his or her encryption scheme via their 401 script that sets a cookie that is only valid to try passwords once. This will make it impossible to guess a user name-password combination in a single connection attempt. (At least impossible in the

sense that if the encryption scheme is any good this is just about impossible.) Let's just assume that we are not totally incompetent and have made certain our users do not have usernames like "aaa" with the same password, okay?

(Note: It is worth mentioning that even if we do allow a small time period of five minutes, and let's say the cracker has 30 hosts to try this from, if we flag at or beyond 100 tries in 5 minutes, we are talking about 2970 requests that can be made before action can be taken. And then they can continue password attempts after that 5 minute time period. That is a lot, in reality. It is better to keep this number small.

Do users really need to try their password 100 times before they realize it isn't going to work? E\*Trade lets their users try three or four times before they have to call in. I think somewhere around 10 is probably enough to sufficiently know that the username/password combinations isn't right, although this will be largely based on your users' needs. There is a valid anomaly in that sometimes a user who has downloaded a page full of links to images inside a secure area will try to go to the page. They then have to hit cancel a few dozen times to stop the htaccess password prompt while their browser attempts to download all the images. So it is wise to have a caveat where null username/password pairs do not count against a user, or be perfect and never have fully qualify paths for your images. Both are probably good ideas.

One might now say that we have only obfuscated the problem, and instead of simply accepting a POST submission or straight Basic-Auth type authentication we are now essentially relying on the strength of the token. Well if you take what I have said on face value, you are right. But there is a way to make this so it is also very hardened from a cracker's perspective. Keeping a local copy of the keys that have been validated via POSTing from your 401 program's login page will not only solve this problem (because there will be no record of the transaction if they simply try brute-forcing the key) but will also be worth a lot when it comes time to do audits. This is especially true if you have functions like an admin-user switching function. It's imperative that you know who switched users to whom.

(Note: This will not stop people from trying. And there is the standard by-hand brute force method, that can be done by going from machine in cyber-cafes to machines in public libraries, etc. There is no way to stop a public web site from being able to be attacked one user name at a time. But it will stop the most blatant abuses, and it will give you a leg to stand on. I would suggest using a one-time-pad token ID in conjunction with htaccess hardening if you are especially worried of this possibility. There is a one-time pad theory that will combine a

username and password with a token given to you from a key-card. Only a single token is usable at any time, and it only can be used once. There is a lot more OTP technology out there. I suggest looking it up for more details.)

Problem solved. So how do we stop crackers who write these complicated programs from continuously hitting the site to test if impossible really means impossible?

### **Worst Case Scenario**

Before I answer that question, I want to make sure that users are aware of the facts. In the discussion above we described the best case scenario (amazing, I know.) How about the worst case? The worst case - and yes, I have seen this first hand - is not just one server, or two, or five, but over 20 different proxy servers connecting to a single host at once.

Yes, you guessed it, a single cracker using a lot of proxies and a little determination to completely bring the web-server to its knees. In some cases we would be dealing with as many as five or more crackers at a time attacking us. This overwhelmed many of our first attempts to stop them, and caused tens of thousands of dollars in down-time, lost customers, and programmer's pay to fix the problem. The net gain for the crackers? Not a single password found and it only cost an hour or so worth of work.

Can you block all proxy servers? Nope. Can you find out who the original host is? Not likely(not that that would help you much, seeing as how he is using a proxy server.) So what can you do? Block the proxy servers that are attacking you for an appropriate amount of time. An inelegant solution for sure, but there isn't a good solution for this available.

### **Morris' Attack**

Another attack worth mentioning, called the Morris attack (because it was pioneered by Robert T. Morris in 1985) is an attack against the ISN (Initial Sequence Number). It was first introduced in a paper called "[A Weakness in the 4.2BSD UNIX TCP/IP Software](#)".

So how does this attack work? Consider HTTP itself. It is a stateless connection working on the application layer, and relies on TCP to negotiate IP. The three-way handshake works as follows. A SYN (synchronize) packet is sent with the client's ISN (we'll call this ISNa) to the server. The server responds with a SYN with the ISN for the server (we'll call this ISNb) and an ACK with

ISNa in it. Then the client responds with an ACK (acknowledge) ISNb and the three-way handshake is complete.

If the return IP address in the initial SYN packet is spoofed, the return packet will be sent (along with the ISNb) to that spoofed address and we will never see it. But it is possible to guess ISNb as sometimes it is a static number, or it is incremented over time, or incremented by the connections made to it. If this number can be predicted a half-duplex connection can be made to the server with a spoofed IP address. That is, packets can be sent to impersonate an IP address, but no return packets can be seen.

Morris' attack might seem on the surface to be a difficult vulnerability to exploit, but there are many situations where IP addresses are used as a form of authentication. Also, it is worth noting that this does not require any sniffing, or man-in-the middle attacks, and leaves no log-files with your IP address in them, which makes it preferable and/or possible to exploit in ways that other TCP/IP attacks might not.

This can be used in something as simple as a configuration in an htaccess file:

```
order deny,allow  
  
deny from all  
  
allow from 123.456.789.123
```

This is a good example of a hole that could be exploited using Morris' attack, but only if a CGI script - that could do something like add a password - was located inside this directory. This would require some amount of inside information on the directory structure and the CGI itself. A better example: a lot of modern credit card processors for internet transactions that write to .htpasswd files do so by a CGI script that uses only the IP address of the user to authenticate.

The best way to get around this is to prenegotiate some form of authentication with the server, as I described in htaccess hardening. In this way we do not have to rely on spoofable IP-based authentication or sniffable Basic-type authentication. Moreover we now have an authentication that is still capable of accepting spoofed packets, but cannot fall prey to a simple half-duplex TCP ISN attack.

More information on Morris' attack, as well as TCP/IP spoofing and ISN prediction can be found at [http://www.kb.cert.org/vuls/id/498440](#).

## Blocking

There are two different potent choices at this point. Firstly, instead of using `mod_auth_external`, write a `mod_PERL` program or an Apache module, or something similar to intercept the data, and close the connection, denying access to that IP address before any authentication takes place, but instead allowing them access to a "reinstate your password."

I have run into some issues with `mod_PERL` regarding IPC. If the `mod_PERL` program dies for any reason or cannot communicate to the parent the whole system will break. The result being that the system will hang and Apache will need to be stopped, while each individual semaphore and chunk of shared memory is removed prior to restart. There are certainly some ways around this, but it is a little more complicated than using `mod_auth_external`. Using Apache modules is a nice way to subvert this problem while getting the speed gains involved in inline authentication, as Apache manages resources for you, and will not let a single child bring the server down. If speed concerns are enough for you to ignore using Fast-CGI (CGIs stored in memory to boost performance) `mod_PERL` might also cause too much overhead for you as well.

Another avenue is to connect your program to some utility like `ipchains` or a hardware firewall to block all packets to your web-server from the cracker's IP address. This is fraught with risks (AOL reuses only a handful of IP addresses for all their millions of users, so blocking users in this way could lead to blocking many more than just a single user.) These are simply some options. You will have to assess the dangers of and the proper reactions to crackers trying to get into your system.

Another problem with hooking directly into `ipchains` without any intelligence involves Morris' attack. It would be possible for an outsider to block a considerable amount of traffic by sending requests that will, to an IDS, or to your program, look like an IP address is performing a brute-force attack. `ipchains` will therefore block that traffic, and cause a Denial of Service against the spoofed host.

## Alternatives

One alternative I have heard to `htaccess` hardening is using `htaccess'` built in digest based

authentication. This is a very robust method of sending the password pair more securely. It works on a one-way hash based off the username/password pair as well as a singular token that can only be used once. This is essentially a very nice way to get around having to use one-time pads. It is not supported by all browsers, but is very common, and cannot be used in replay attacks, passwords are not susceptible to packet-sniffing. It uses a 128 bit digest, and is considered computationally un-brute-forceable.

Okay, you can stop people from using the majority of cracking tools to break into your password protected directory. Out of the box that seems like a great idea, except for one thing. Regardless of if crackers can break in or not, it will not stop them from trying. If they still get a 401 error, or if the text is different for the error as the valid directory, this solves little. They will still bring your server to a halt, and will still cause your customers to complain. It is worth mentioning digest-based authentication because it is a good method, and has strong advantages to normal Basic-Auth type authentication. If you must use htaccess and cannot afford the time to make any of the changes suggested in this paper, use digest-based authentication.

A last note on this particular method. While it is still possible to perform a brute-force attack, it is not likely to ever find a real-password, and therefore I haven't seen any tools written that try to brute force digest based authentication. This means that for the time being, this method is much safer in the sense that a tool like wwwhack will not hurt system resources as a result of trying to break it, only because no one has yet tried to break digest based authentication in this way. More information on Digest type authentication can be found [here](#).

Another alternative that has been brought to my attention circumvents the need for much of the custom programming involved in htaccess hardening. A single proxy type server could act as a gateway between the client and the server. It would intercept each request and decide what to do with it, and take each answer and change headers, place cookies, etc... to provide the same authentication method, without the hassles of messing with configuring the server to work with htaccess hardening.

There are several benefits gained by the proxy method. Firstly, and most importantly, this method now provides this type of authentication system to web-servers that could not ordinarily work with it (including the Windows platform). Furthermore, a single proxy could be used for multiple servers and would be easier to administrate. The only prerequisite is that the web-server behind the proxy would need to return 401 headers (or some other mechanism to

tell the proxy that it is looking at a protected directory) continuously as to tell the proxy to use it's built in authentication system.

Another benefit of the proxy method is that it can connect directly into an IDS (Intrusion Detection System) and could provide the components of a firewall via IPChains or some packet redirection mechanism.

This alternative is a theory, more than a mechanism that I have spent any time creating, and might have unforeseen problems that I have not taken into account. I suggest taking this method with serious caution as it is unexplored territory.

## **Conclusion**

We have proven that with not that much effort administrators can harden htaccess making it a very robust solution. I cannot stress more that you should download a copy of wwwhack, and/or write a client to connect to web servers that allows you to see and print headers to the web server. All these things will help when it comes time to deploy your web server. Knowing the symptoms of an attack can only help you. Administrators can be aware of the risks and, more importantly, develop an effective reaction strategy.

## **Thanks**

Special thanks go to Ethan Brooks, Goose, Vacuum, Bronc Buster, John Stewart, Ken Williams and Ali Nazar for their help in contributing to and/or editing this paper. Thanks to RLoxley, Teeceep and Peter Shipley for letting me bounce these ideas off them, and thanks to Robert Morris for putting up with my admittedly rather impromptu phone call.

[Privacy Statement](#)

Copyright 2006, SecurityFocus