The problem with cookie authentication systems

By Cal Henderson

2009-05-11: This problem has been given a name in the time since I wrote this article, back in the dark ages. We call it Cross Site Request Forgery, or CSRF for short. This solution outlined below is now standard practice across the web and is generally known as using a 'secure crumb'.

Many web based systems use cookies for authentication. Some store the current user's username and password. Some "more secure" systems store a session id, the password having been passed over in a HTTPS POST for extra security. Some systems just store a user id in a cookie (and hope for the best). But this is all irrelevant - the point being that the details necessary to authenticate a user for each request are stored in a cookie, one that is sent to the server on every request.

Imagine the following page, simplified for brevity:

```
<?php
 if ($done){
  if (athenticate_cookies()){
   do_some_admin_feature();
   echo "done!";
  }else{
   echo "bad user!";
  }
 }else{
?>

<form action="self.php" method="post">
<input type="hidden" name="done" value="1">
<input type="submit" value="Do Something">
</form>

<?php
 }
?>
```

Now, when an unauthenticated user presses the button, they get the "bad user" message. When an authenticated user presses the button, the function is performed and they get the "done!" message. Great, a perfectly secure. Well, not quite. What if the unauthenticated user tricks the authenticated user into following the following link:

```
http://www.foo.com/admin/self.php?done=1
```

Perhaps the authenticated user doesn't read the url before clicking. Perhaps the url is munged or put through a url shortening service. Or perhaps, far more cunningly, it's put as an <img> src attribute on a messageboard.

The authenticated user follows the url and the function is performed. In the case of the image trick, the authenticated user doesn't even know they've done it.

So how can we solve it? The naive fix is to change the code as follows:

```
if ($HTTP_POST_VARS['done']){
```

This means that a GET request (with ?done=1 on the end of the url) will not work. But this is trivially

easy for a hacker to get around. All they have to do is write a script (in PHP perhaps) that performs a HTTP POST. Infact, now they've gone to the trouble, they can easily disguise it too. They write a PHP script to do it, and call it foo.gif. They then set foo.gif to be parsed as PHP, by setting an alias in their .htaccess file. As a final touch of genius they make the PHP script output an image after doing the HTTP POST. So now it looks like an image url, acts like an image url, but allows the hacker to perform any administration action they want.

This isn't just a scare story. Don't believe it will never happen. It's happend to me.

So how can we battle this? It's actually pretty simple, but takes a little extra work for every authentication. All you have to do, is take the authentication cookie (if you are using username and password cookies, use the password) and turn it into a POST variable too. Once the form is submitted, check the cookie against the post variable:

```
if ($HTTP_POST_VARS['session_check'] != $HTTP_COOKIE_VARS['session']){
 echo "authentication failed!";
}
```

and...

```
<input type="hidden" name="session_check"
 value="<?php echo htmlentities($HTTP_COOKIE_VARS['session']); ?>">
```

This ensures that a value unknown to the hacker (a password or session id) must be passed along for each request, independant of the cookie.

This is only one step in ensuring web applications are secure, but it's an important one. It's all very well spending time and money on a fantastic authentication system, just for a hacker to come along and trick authenticated users into doing their bidding.

(end of article)