blog.codinghorror.com

Cross-Site Request Forgeries and You

6-7 minutes

23 Sep 2008

As the web becomes more and more pervasive, so do web-based security vulnerabilities. I talked a little bit about the most common web vulnerability, cross-site scripting, in Protecting Your Cookies: HttpOnly. Although XSS is incredibly dangerous, it's a fairly straightforward exploit to understand. **Do not allow users to insert arbitrary HTML on your site.** The name of the XSS game is sanitizing user input. If you stick to a whitelist based approach -- *only* allow input that you know to be good, and *immediately* discard anything else -- then you're usually well on your way to solving any XSS problems you might have.

I thought we had our website vulnerabilies licked with XSS. I was wrong. Steve Sanderson explains:

> **Since XSS gets all the limelight, few developers pay much attention to another form of attack that's equally destructive and potentially far easier to exploit.** Your application can be vulnerable to cross-site request forgery (CSRF) attacks not because you the developer did something wrong (as in, failing to encode outputs leads to XSS), but simply because of how the

whole Web is designed to work. Scary!

It turns out I didn't understand how cross-site request forgery, also known as XSRF or CSRF, works. It's not complicated, necessarily, but it's more.. subtle.. than XSS.

Let's say we allow users to post images on our forum. What if one of our users posted this image?

```
<img src="http://foo.com/logout">
```

Not really an image, true, but it will force the target URL to be retrieved by any random user who happens to browse that page -- **using their browser credentials!** From the webserver's perspective, there is no difference whatsoever between a real user initiated browser request and the above image URL retrieval.

If our logout page was a simple HTTP GET that required no confirmation, **every user who visited that page would immediately be logged out.** That's XSRF in action. Not necessarily dangerous, but annoying. Not too difficult to envision much more destructive versions of this technique, is it?

There are two obvious ways around this sort of basic XSRF attack:

1. Use a HTTP POST form submission for logout, not a garden variety HTTP GET.
2. Make the user confirm the logout.

Easy fix, right? We probably should never have never done either of these things in the first place. Duh!

Not so fast. Even with both of the above fixes, you are *still*

vulnerable to XSRF attacks. Let's say I took my own advice, and converted the logout form to a HTTP POST, with a big button titled "Log Me Out" confirming the action. What's to stop a malicious user from placing a form like this on their own website ..

```
<body
onload="document.getElementById('f').submit()">
<form id="f" action="http://foo.com/logout"
method="post">
<input name="Log Me Out" value="Log Me Out" />
</form>
</body>
```

.. and then **convincing other users to click on it?**

Remember, the browser will happily act on this request, submitting this form along with all necessary cookies and credentials directly to your website. Blam. Logged out. Exactly as if they had clicked on the "Log Me Out" button themselves.

Sure, it takes a tiny bit more social engineering to convince users to visit some random web page, but it's not much. And the possibilities for attack are enormous: with XSRF, **malicious users can initiate any arbitrary action they like on a target website**. All they need to do is trick unwary users of *your* website -- who already have a validated user session cookie stored in their browser -- into clicking on *their* links.

So what can we do to protect our websites from these kinds of cross site request forgeries?

1. **Check the referrer**. The HTTP referrer, or HTTP "referer" as it is

now permanently misspelled, should always come from your own domain. You could reject any form posts from alien referrers. However, this is risky, as some corporate proxies strip the referrer from all HTTP requests as an anonymization feature. You would end up potentially blocking legitimate users. Furthermore, spoofing the referrer value is extremely easy. All in all, a waste of time. Don't even bother with referrer checks.

2. **Secret hidden form value**. Send down a unique server form value with each form -- typically tied to the user session -- and validate that you get the same value back in the form post. The attacker can't simply scrape your remote form as the target user through JavaScript, thanks to same-domain request limits in the `XmlHttpRequest` function.

3. **Double submitted cookies**. It's sort of ironic, but another way to prevent XSRF, essentially a cookie-based exploit, is to *add more cookies!* Double submitting means sending the cookie both ways in every form request: first as a traditional header value, and again as a form value -- read via JavaScript and inserted. The trick here is that remote `XmlHttpRequest` calls can't read cookies. If either of the values don't match, discard the input as spoofed. The only downside to this approach is that it does require your users to have JavaScript enabled, otherwise their own form submissions will be rejected.

   If your web site is vulnerable to XSRF, you're in good company. Digg, GMail, and Wikipedia have all been successfully attacked this way before.

   Maybe you're already protected from XSRF. Some web frameworks provide built in protection for XSRF attacks, usually

through unique form tokens. But do you know for sure? Don't make the same mistake I did! Understand how XSRF works and ensure you're protected *before* it becomes a problem.