weblinks

Cross Site Request Forgery (CSRF/XSRF)

4TH JUNE, 2010

If you're building a site that allows users to update any sort of information (so most websites), then you should probably think about protecting against Cross Site Request Forgery (referred to as CSRF or XSRF). Being susceptible to this type of attack can be annoying in some cases, but extremely dangerous in others. Unfortunately, it's not the type of attack that's easy to understand at first, and it's not immediately obvious how to prevent such an attack. Because of this, protecting against XSRF is often overlooked, even on some big name websites.

What is XSRF?

An XSRF vulnerability is one which allows a malicious user (or website) to make an unsuspecting user perform an action on your site which they didn't want to happen.

As a basic example, imagine you allow users to post images in your comments. If a malicious user puts *"http://example.com/logout.php"* as the image's URL, where example.com is your domain, then any time a logged in user views that comment they will be logged out if you don't protect against XSRF. It's not a valid URL for the image, but that doesn't matter as the unsuspecting user's browser will still make the request and your site will perform the action thinking the user wanted it.

A more dangerous example could be that you allow a user account to be deleted without confirming the action or protecting it from XSRF in any way, so any user visiting the page would then get their account deleted instantly!

> *I thought I'd make a comment on your site. Check out this cool image!*
> **

Even if you have no user submitted content on your site, you can still be vulnerable. If a malicious website contains a form which the unsuspecting user submits, it can submit the information to a different site which the user is logged in to, and that site will think the request came from the logged in user and will just perform the update as if the user had done it intentionally.

It doesn't even have to be a link/button the user clicks on, as shown in the first example it could happen even if just viewing a site. Obviously, this could become quite annoying for your users.

While annoying is one thing, it can also be dangerous. For example, let's say I'm logged into an account on a simple shopping cart site. I then go and browse to another unrelated website. The other website has a button which just says "Click here to register". Seems simple enough, this other website wants me to register an account. However, this is a malicious site and when I click the link, it's actually submitting a

request to the original shopping cart website as if I'd clicked the "Order me 1000 of some expensive item in one-click" button. If the shopping cart website is susceptible to XSRF then it will think the request to order 1000 items was genuinely submitted by me, and I'll get a nice surprise in the mail and on my credit card statements.

It's a particularly difficult type of attack to get your head around as it's very subtle, but once you understand how it works it's not that difficult to protect against it. It's very easy to go down the bad route though and think you're safe when in reality you're still wide open to attack.

The Wrong Way To Protect Against XSRF

So what can you do as a web developer to prevent such attacks? Effectively you're just wanting to make sure a request came from your site and was actually intended to be run by the user, so you could just check the referer header to make sure the request came from your own site, right?

```
<?php
$url = parse_url($_SERVER['HTTP_REFERER']);
if ($url['host'] != "wblinks.com")
{
    die("You're not coming from my site. Possible XSRF attack");
}
```

While this may work in some cases, it's going to be about as effective as an underwater hair dryer as far as stopping an XSRF attack. Altering the referer header is pretty trivial and it will also have the downside of making the site unusable for lots of people, since many browser or proxies can strip the referer header when in "private" mode.

This also wouldn't protect against the first example of XSRF, where someone just uses the logout URL as an image URL in a comment. As the request would come from the correct referer in that case.

So not only will you not prevent XSRF attacks, but you'll also annoy some of your users. Not a good solution.

The Right Way To Protect Against XSRF

What we really need is a nonce (https://en.wikipedia.org/wiki/Cryptographic_nonce) (one-time key/token) which allows us to validate that the request came from a form we presented to the user intentionally. The following code samples show the method I use to achieve this for the comments section of this site.

For the purposes of the following code examples, you can assume the **Session** class is just a wrapper for the $_SESSION superglobal (It actually does some other stuff, but that doesn't matter for this). All of these functions are part of an **Auth** class, which handles all of my XSRF protection (along with some other things).

First things first, you'll want to add a configuration parameter somewhere so you can specify how long the tokens should last. I use 15 minutes, but you can use whatever makes sense for your application. The important point is that the tokens should have an expiry,

```
$_config['csrf_token_lifetime'] = "15 mins"; // How long the tokens should
last.
```

You'll also need a function which can generate a nonce (basically a random string of characters), here's the method I currently use, there might be better methods out there.

```
// Generates a nonce, by base64 encoding some random binary data.
public static function generateNonce($length = 20)
{
    return substr(base64_encode(openssl_random_pseudo_bytes(1000)), 0,
$length);
}
```

To prepare the token, you'll want to generate it, calculate the expiry time for it (based on the configuration parameter you stored earlier), and then add these values to the current user's session. I keep an array of active tokens (more on why later).

```
// Generate a new XSRF token and store it in the user's session.
public static function getXSRFToken()
{
    $nonce           = Auth::generateNonce();
    $tokens          = Session::get("_xsrf");
    $tokens[$nonce] = strtotime("+".$_config['csrf_token_lifetime']);
    Session::set("_xsrf", $tokens);
    return $nonce;
}
```

You'll need some way to validate that a given token is valid. This is something you'll call when you recieve a token as part of a request. Validate that it exists in the user's session, and that it hasn't expired. If it's valid, you should make sure to immediately invalidate it so that it can't be used again. You also want to clear any tokens which have expired.

```
// This will determine if an fkey is valid
private static function validateXSRFToken($xsrf)
{
    $tokens = Session::get("_xsrf");
    if ($tokens == null) { return false; } // Sanity check

    // Check that the fkey exists, and time has not expired
```

```
    foreach ($tokens as $key => $expires)
    {
        // Remove any tokens that have expired.
        if (time() > $expires)
        {
            unset($tokens[$key]); Session::set("_xsrf", $tokens);
            continue;
        }

        // If key matches and isn't expired, we can use it.
        if ($key == $xsrf  && time() <= $expires)
        {
            // Key is good, remove it from use
            unset($tokens[$key]); Session::set("_xsrf", $tokens);
            return true;
        }
    }
    return false;
}
```

Rather than having to call the *getXSRFToken()* function manually on every form on your site, you should probably make a quick helper function to do it for you. This will also make sure you don't typo the name of the field anywhere.

```
// Creates the form input to use
public static function getXSRFFormInput()
{
    return "<input type=\"hidden\" name=\"fkey\"
value=\"".Auth::getXSRFToken()."\" />";
}
```

So what's going on with all this? Anytime there's a form on the site which POST's data, I will output the **getXSRFFormInput()** function to add a hidden field to this form. This hidden field contains the value returned from **getXSRFToken()**, this method generates a random 20 character nonce, and stores it into the user's session and then returns this nonce so it can be put into the hidden input. For example,

```
<form action="do_something.php" method="POST">
    <fieldset>
        <?php echo Auth::getXSRFFormInput(); ?>
        <input ... />
    </fieldset>
</form>
```

This will add a hidden field to the HTML form which will look like this,

```
<input type="hidden" name="fkey" value="3748ab53cf129d536eca" />";
```

The user's session will also now contain an array called "_xsrf", which contains a random 20 character nonce, along with the expiry time of that nonce.

```
$_SESSION['_xsrf'] array(1) =>
{
    ["3748ab53cf129d536eca"] => int(1275675864)
}
```

When the user submits this form, the idea is that we take the fkey value and check it against the ones we've stored in the user's session. If it's there and hasn't expired, then the request is valid and came from a form which the site generated. If it doesn't exist in the session, or has expired, then it's a possible XSRF attack and the output should be stopped and logged. This validation is done in the **validateXSRFToken()** method.

Now the final step, is to run something like the following code on any page that takes input. I actually have it in my bootstrap file which is run on every page request to the site.

```
if (count($_POST) > 0)
{
    if (!Auth::validateXSRFToken($_POST["fkey"]))
    {
        Log::create("XSRF", "Possible XSRF attack", LOG::NOTIFY,
LOG::NOTIFY_ADMIN);

        // Inform user that why things broke and how to fix it.
        Notification::set("Your session token expired. Please refresh and try
again (don't use the back button).", NotificationType::ERROR);

        // Redirect back to whichever page they came from.
        header("Location:".Session::getReferer()); exit();
    }
}
```

So if anyone ever makes a POST request to the page, we not only validate that the fkey exists in the user's session, but that it also hasn't expired. Restricting the validity time of the fkey reduces the likelyhood of an attack succeeding since it would have to be mounted quickly. Rather than purely relying on the expiry time, we also invalidate any key as soon as it's used, so that it can't be used again if it was captured by a MITM attack of some sort.

Why an Array of Tokens?

It was just the choice I made when first implementing this, it means each form gets it's own token value. This allows users to have another browser tab open and be able to submit both without getting an error (if I used the same token, it would be invalidated after the first form is submitted). If that's not a use case you need to support, then you could just store one XSRF token in the user's session and use it on every form on a page. Then just cycle the token once it's used. This should be just as effective as my method above where I generate a new token for each individual form. The important things to remember are,

> 1. The token must be tied to a specific users' session, it should not be site-wide!
> 2. The token should have some sort of expiry time.
> 3. The token must be invalidated once it's been used.

Why only POST?

You may be wondering why I'm only checking POST variables to prevent XSRF rather than both POST and GET. The answer is because GET should never be susceptible to such an attack if you're using it correctly.

So when should you use GET and when should you use POST? Well, it's all in the name. GET requests ideally should be used when the contents of the page are read-only, so nothing gets changed by the request. So a GET request should be idempotent (https://en.wikipedia.org/wiki/Idempotence) (I should be able to trigger the same GET request as many times as I want and it shouldn't affect the result I get. Basically, you want to make sure it doesn't have any side-effects). POST should be used whenever it causes a destructive action (I don't just mean deletes... I mean destructive as in something changes). So login, logout, updates, creation, deletion, comments, voting, etc.

This is why browsers will generally prompt you to confirm when resending a POST request, whereas they won't with a GET request. This is because a POST request will generally change the outcome each time it is run so you want to make sure you're not going to accidentally run it again and change something you didn't want to. A GET request shouldn't change any data that way, and so it doesn't need to be confirmed.

If you do want to use a GET for a destructive action, say you want it to be an anchor tag rather than a button otherwise your style will look strange, then you must make sure that you at least redirect to a confirmation page to confirm the YES/NO, ideally this confirmation page should use POST.

If not, then say you were to have a link in your admin area which deleted an item using GET, without any form of confirmation. Some browsers do what's called pre-fetching, where they examine the links on a page, and pre-fetch the websites you'd get by cliking these links, under the assumption that you will go to them. Then when you do click them, the page can be displayed very quickly. If you just delete with a GET, then simple visiting the admin page could cause the browser to follow all those links in the background and delete everything. Obviously not something you want. Yes, this happened to me in the past, so please learn from my mistakes. (I've heard stories of GoogleBot accidentally deleting pages from a site this way too).

POST Refresh

Another thing to keep in mind, is that you should redirect users to a GET based URL once the POST request has been handled. This means users will be able to hit F5 and refresh the result page without being prompted if they want to resubmit the POST request (since that will now cause an XSRF error to be displayed). This is common practice on most sites and something users now expect to be able to do. So they'll probably not be too happy if they have to re-send requests and then get errors about their XSRF token expiring.

Final Thoughts

As I have said many times before, **I am not a security expert**. Take everything you've read here with a grain of salt. This is just how I understand things right now. I may have misunderstood something and there could be a glaring security hole in my examples above (please do let me know if that's the case!). Things might also change in future, new attack vectors may be discovered, or better ways of protecting against XSRF may become standard. You're highly encouraged to do your own research before settling on a method of XSRF protection.

Additional Reading

Other articles/posts on similar subject matter (some of these may be more recent than this one),

i. OWASP Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet (https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet)

References

A list of all the links that appear in this note,

i. https://en.wikipedia.org/wiki/Cryptographic_nonce
ii. https://en.wikipedia.org/wiki/Idempotence
