Cross-site scripting request forgeries

ATTACK OF THE CSRF

Sometimes, even ING, YouTube, The New York Times, and Google get it wrong. **BY KURT SEIFRIED**

Cross-Site Request Forgery (also referred to as Cross-Site Reference Forgery, CSRF and XSRF) is apidly becoming a serious security problem of which most programmers and users are blissfully unaware. CSRF is a web-based attack that has grown out of, and remains a close cousin to, the more traditional Cross-Site Scripting (XSS) attacks. In an XSS exploit, the attacker inputs malicious content into a web application (e.g., by creating a malformed URL or embedding hostile code in a response box) that results in hostile content such as JavaScript being inserted into otherwise safe content that then is served to the victim. CSRF attacks take it a step further by inserting hostile content that results in an action by the user's web browser, such as changing a filter setting within web-based email or initiating a money transfer from an online bank account.

A CSRF Attack Example

So you go to your favorite social networking site to chat with friends. Unfortunately, the site in question allows users to insert images into web-based conversations (e.g., avatars for a forum). Instead of using a URL such as:

```
<img src="http://random-site/
image.jpg">
```

The attacker uses a URL such as:

```
<img src="http://social-site/
changepassword?newpassword=
password000">
```

Thus when a user's web browser attempts to load the image it instead connects to the social networking site and executes a command to change the password.

This attack can also be leveraged from other sites. For example, if a user remains logged into the social network site while browsing the web in another tab and an image on another site points to the change password URL, that tab would execute the command, and unless the site had specific CSRF protections in place, the user's password would end up being changed.

CSRF attacks have become popular for three simple reasons. The first is the emergence of web-based services such as email, online commerce, banking, and so on. CSRF attacks can result in money being sent to an attacker through a web-based bank or stock trading site. Web-based email allows an attacker to reset or request copies of your passwords for various services such as DNS registrars and online commerce sites. Attackers can monetize these attacks by directing access to bank accounts, resetting a user's password, and so on. Something as simple as resetting a password can result in an attacker holding a user's account, domain, or service hostage. For a small fee, the attacker will reset the password and return it to the user (see Figure 1).

The second reason is the presence of tabbed browsing. When web browsers first came out, browsing the web was a largely serial experience. It didn't occur to me for some time that I would ever need more than one session because the content wasn't such that I wanted to stay with it (web browsing was quite literally web browsing). However, with the advent of web-based email, I now have three sessions just sitting logged in so I can send email quickly and be notified when new email comes in. This means that a CSRF attack is much more likely to succeed because I am always logged in to my web-based email (I use a separate browser for my email to prevent exactly this).

The third reason is that most web applications have no security. They are absolutely terrible at filtering user input properly, allowing attackers to inject malicious content (such as JavaScript) via any number of cross-site scripting vulnerabilities. Although I rarely visit hostile websites, I visit a lot of "trusted" websites that I know for a fact have poor filtering that can result in XSS attacks, ultimately allowing for CSRF attacks.

Additionally, few web applications implement CSRF protections that will prevent such attacks.

Defenses for Programmers

The way to beat CSRF is simple in concept, and depending on your web-based application, anywhere from easy to almost impossible to implement properly. To beat CSRF attacks, an application simply has to verify that each request is made properly; in other words, your application needs to maintain state so that the content in browser tab 'A' that is logged into your web-based email is the only one allowed to send email, and a request made via content in browser tab 'B' that is accessing a hostile website does not result in email being sent or read. To do this, your web application has to maintain state information. But the web was designed as a stateless system from the very beginning, so any addition of state is going to require some technical trickery because the web browser itself can't help you directly.

To link content in a web page to a request made from that web page properly (i.e., the user fills out a form and hits *Submit*), you need to pass a one-time token in the content that the web browser then passes back with the request, allowing you to confirm that the request came from the right place.

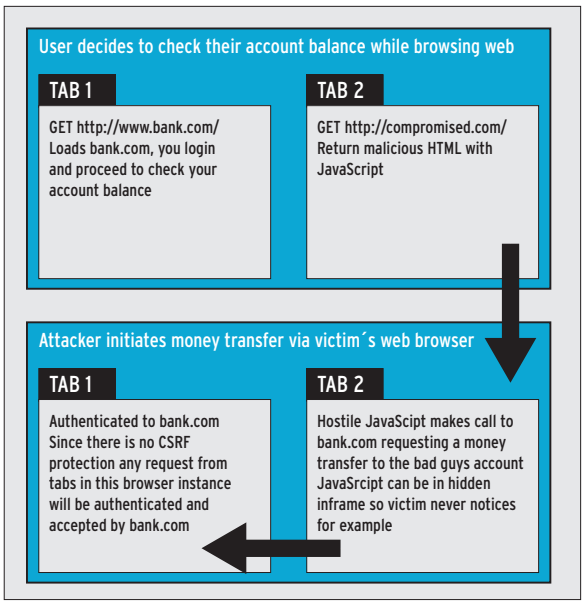

Figure 1: Example of how the CSRF attack works.

Send and Receive Tokens

Now this one-time token can be sent and received in a number of ways.

- Hidden Form Fields

```
<input type="hidden" name="token" value="randomstring" />
```

The advantage here is that many applications support adding form fields and logic to process them. The disadvantage is that web pages that don't use forms but still allow interaction can't be addressed as easily by this technique.

- URL Components (Within Either the URL or Parameters)

```
http://example.org/newpassword?new=password&token=randomstring
```

This one has the advantage of making the data available to the server, so you could, in theory, have an Apache module that validates all requests and simply blocks any invalid ones, preventing the application from ever seeing them. The disadvantage (or potentially an advantage depending on your point of view) is that users can no longer bookmark a page because the one-time token will no longer be valid.

- Cookies

```
PHP: setcookie("TokenCookie", $randomstring);
```

Cookies must be enabled for this to work, and potentially they can be stolen by a clever attacker (various browser flaws have allowed for cookie stealing over the years). The advantage of this technique however is that it is largely invisible to the user and does not require either that HTML be displayed to the user or that the URL to be used be modified in any way.

- Requirements of the Back End

All of these examples require some form of back end to store the session data and create session tokens, compare them, and allow or disallow requests based on them. Additionally, web-based applications might need to be modified (e.g., if hidden form fields are used to pass the data). The good news is that more and more web applications are implementing this protection by default. For example, the popular Joomla! Framework has the *JRequest::checkToken()* function now.

Defenses for Web Users

The good news is that a number of defenses against CSRF attacks are available for web browsers. A common one is the NoScript plugin for Firefox. Unfortunately, for NoScript to be effective, you need to disable JavaScript by default and then selectively enable JavaScript for sites you trust. This leads to obvious usability issues because many sites do not work at all or very poorly if JavaScript is not enabled. Additionally, it will not prevent an attacker from leveraging a cross-site scripting flaw in a site you trust.

However, not all browsers support such selective control over which sites get to execute JavaScript. Another option is simply to install a separate web browser or run a separate instance of a web browser and use it for trusted online activities such as web-based banking and email.

One browser that has incorporated this strategy is Google Chrome. Each browser tab in Chrome is actually a separate process and not a thread running within the same context as other threads (tabs). Thus, the tabs cannot interfere with each other, rendering most CSRF attacks impotent. (To be attacked successfully, you would have to log in to a web-based service, then using that same tab, go to a hostile site.) ■

INFO

[1] Cross-Site Request Forgery (CSRF): *http://www.owasp.org/index.php/Cross-Site_Request_Forgery*

[2] Zeller, W., and Felten, E.W. "Cross-Site Request Forgeries: Exploitation and Prevention," 2008, *http://www.freedom-to-tinker.com/sites/default/files/csrf.pdf*

THE AUTHOR

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.