White Paper

CSRF: Attack and Defense

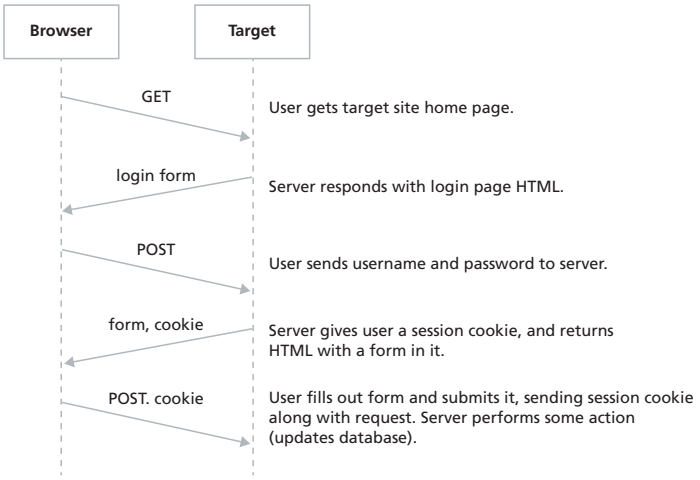
By Jeremiah Blatz
Managing Consultant
McAfee® Foundstone® Professional Services

Table of Contents

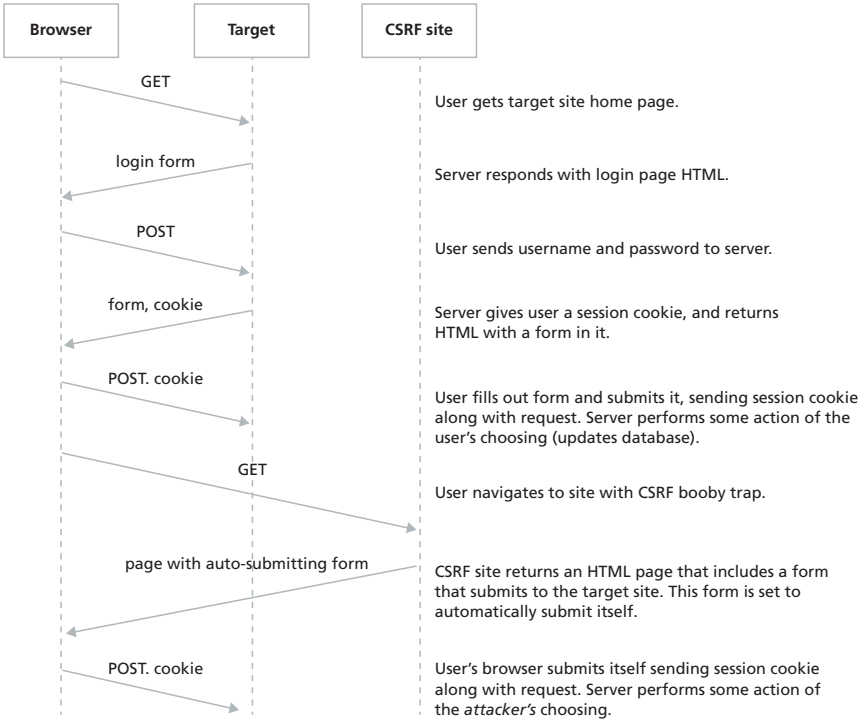
Definition of CSRF

CSRF stands for cross-site request forgery. It's also known as session riding or XSRF. CSRF takes advantage of the inherent statelessness of the web to simulate user actions on one website (the target site) from another website (the attacking site). Typically, CSRF will be used to perform actions of the attacker's choosing using the victim's authenticated session. If a victim has logged into the target site, an attacker can coerce the victim's browser to perform actions on the target website.

To go into more detail, let's look at what happens when a user visits a website:



Here's what happens in a CSRF attack:



From the point of view of the web server, there is nothing unusual about the second POST request. It's well formed, and contains a valid session cookie. Naively, the web server will process the request.

Here are some important properties of CSRF:

- *The victim need not be "logged in," depending on the attacker's goals*—While the most common goal of CSRF is to exploit the victim's authentication to perform some authenticated action, CSRF can be used for a variety of attacks. For example, an attacker might use CSRF to perform fraudulent, unauthenticated actions, such as voting in an online poll. Or, an attacker might launch a denial-of-service (DoS) attack using CSRF, by posting a message on a popular message board that made demanding requests to another site. Lastly, CSRF can be used to abuse an insecure trust relationship between a browser and web server, such as a website that uses IP address restrictions, HTTP basic authentication, or SSL client certificates.
- *The attacker can make any request of series of requests on the target site*—In other words, it doesn't matter how many steps the action requires; the attacker can just script multiple requests
- *CSRF works extremely well with other attacks*—While performing valid actions on the target site could be very useful to an attacker, CSRF can also be used with other attacks to create a "death by a thousand cuts" scenario. Some examples include using CSRF to exploit post-authentication cross-site scripting or SQL injection attacks, or to exploit reflected cross-site scripting that can only be exploited via HTTP POST.
- *The attacker cannot usually read data from the target site*—While CSRF is a powerful tool for attackers, it is severely limited. Due to the same origin policy, scripts on one site cannot read data from scripts on another site. Therefore, multi-step actions where the results of one step must be fed into the next step are typically immune from CSRF. There are, of course exceptions to this rule. For example, if the aforementioned outputs are predictable, the attacker may be able to guess or brute-force them. Also, if there is a single cross-site scripting vulnerability on the target site, all bets are off. An attacker may be able to leverage the cross-site scripting to launch read-write attacks against the target. There may be other edge cases where an attacker may be able to read some data off the target site.

Attack Vectors

In these examples, our victim site is target.example.com. It has a page on it, form.html, which sends its data to action.html. In legitimate use, the user will load http://target.example.com/form.html, fill out the data in the form, then hit the submit button. His or her browser will then send a request to http://target.example.com/action.html, which will perform some action using the data.

The goal of the attacker is to cause the victim's browser to submit data of the attacker's choosing to action.html. What are some of the ways the attacker can do this?

Inline "image" links

Creating a CSRF attack can be done in several ways. The simplest is to embed a reference to an external resource (for example, an image) in another page. This attack is very easy to launch, because almost all bulletin board sites allow users to embed images in their posts. Bear in mind that the browser has no way of knowing that the requested "image" is actually a web page that performs some action.

As an example, the following post on a message board would cause the browser to send a request to the target web server:

```
I agree with the above poster
<img src="http://target.example.com/action.html?field1=foo&field2=bar" width="1"
height="1">
```

Auto-submitting forms

If the action requires HTTP POST, the attack is slightly more complex. The attacker can create a form using HTML or JavaScript. These require the attacker to have some degree of control over the CSRF site, as they will have to embed their own HTML in the site. They can gain this control either by being the site owner or by finding some sort of cross-site scripting vulnerability in the site. Unfortunately for us, finding websites with cross-site scripting vulnerabilities is frighteningly easy; the percentage of sites with cross-site scripting vulnerabilities is estimated to be very high—up to 70 to 80 percent.[1]

Here is an example using an HTML form on the attacker's site, say http://attacker.example.com/CSRF.html:

```
<html>
 <body onload="document.frames[0].submit()">
  <form action="http://target.example.com/action.html" method="POST">
   <input name="field1" value="foo">
   <input name="field2" value="bar">
  </form>
 </body>
</html>
```

The attacker would inject the following into the CSRF site:

```
<iframe width="0" height="0" style="visibility: hidden;"
src="http://attacker.example.com/CSRF.html">
```

It is also possible to use cross-site scripting to cause a non-malicious page on the CSRF site to act as a malicious page. In other words, any site on the Internet with a cross-site scripting vulnerability can be used to launch a CSRF attack. This frees the attacker from having to host his or her own attack page.

Phishing

The easiest way to exploit CSRF, from a technical point of view, is to have complete control over the CSRF site and then convince your victim to visit the site. Phishing can be extremely useful for both large-scale and targeted attacks.

For large-scale attacks against consumer banks and the like, the phisher can supplement their information-gathering with actual actions on the target site. Even if the victim balks at giving up their information, they may already be compromised.

For targeted attacks, phishing is even more effective. Intranet and administrative systems are excellent CSRF targets, and an attacker can tailor his or her phishing emails for those targets. For example, if attacking an intranet, phishers can send an email purporting to be from a corporate training partner or insurance provider. If attacking a blog, phishers can email the maintainer about a cool site. If attacking a helpdesk system, phishers can email support about a problem with the site. The victim needn't perform any actions on the CSRF site—merely visiting the site is enough. Additionally, in most of these examples, the victim has a very good chance of being logged in when they receive the phishing email.

Capabilities of CSRF Attacks

As has been noted, CSRF attacks allow an attacker to send HTTP requests to a third-party site using the victim's web browser. That's a pretty wide net. Below are some specific examples. Bear in mind that the target site can be any site that's accessible from the victim's browser. This includes intranet sites and other sites that are not even accessible from the Internet.

Simulate valid requests

When people talk about CSRF attacks, this is what they typically mean. An attacker uses CSRF to fraudulently perform normal actions on the target site. Actions might include:

- Changing a users email address and then performing a "forgot your password" operation to gain access to the user's account
- Adding an account for a user's blog[2] or other system
- Transferring funds from a user's bank account
- Putting in buy orders for a stock as part of a "pump and dump" scheme
- Checking for the existence of a file. This can be used to find and fingerprint web servers on an intranet.[3]

Activate XSS, SQL injection

Beyond normal operations, CSRF can be used to exploit vulnerabilities on the target website. When prioritizing remediation, site owners often de-prioritize cross-site scripting on administrative and intranet systems, XSS attacks that can only be exploited via HTTP post, and SQL injection in parts of the site only accessible to trusted users. With CSRF, all of these are essentially vulnerable to unauthenticated attackers.

Call web services

In some cases, CSRF attacks can call web services. By using an "enctype" attribute of a HTML form, attackers can pass semi-valid XML to a web service endpoint, for example:

```
<form action="http://target.example.com/webservice"
      method="POST" enctype="text/plain">
<input name="<msg><attr><name>a</name><value>b</value></attr></msg>" value="">
<input type="submit">
</form>
```

The one caveat to this is that the attacker cannot set request headers, so requests to SOAP web services (which require a "SOAPAction" header) will not work.

Protecting Your Website

Solutions that don't work

Before launching into a description of how to defend websites against fraudulent requests, first we will discuss some solutions that do not provide adequate protection. The methods discussed below may make an attacker's job slightly more difficult, but will not stop him or her from executing CSRF attacks.

Confirmation screens

It is reported that some people think that requiring confirmation for user actions provides protection against CSRF. Simple confirmation screens do not provide any measure of protection against such attacks. If the form values are stored in hidden fields, the attacker merely needs to send the request that the confirmation screen generates. If the form values are stored in the server session, and the confirmation screen causes the server to actually perform the requested action, then the attacker just has to send two requests with a delay between them. For example, modifying the HTTP POST example above, we can use the following HTML:

http://attacker.example.com/CSRFstep1.html:

```
<html>
 <body onload="document.frames[0].submit()">
  <form action="http://target.example.com/action.html" method="POST">
   <input name="field1" value="foo">
   <input name="field2" value="bar">
  </form>
 </body>
</html>
```

http://attacker.example.com/CSRFstep2.html:

```
<html>
 <body onload="setTimeout('document.forms[0].submit()', 4000)">
  <form action="http://target.example.com/confirm.html" method="POST">
   <input name="confirm" value="true">
  </form>
 </body>
</html>
```

CSRF site:

```
<iframe width="0" height="0" src="http://attacker.example.com/CSRFstep1.html">
<iframe width="0" height="0" src="http://attacker.example.com/CSRFstep2.html">
```

Either way, the application remains vulnerable.

Using POST

One defense against trivial CSRF attacks is just as trivial. Every web script that causes some state change, be it a database insert, writing a file, or moving a robot camera, is supposed to use HTTP POST.[4] This is due to the nature of the underpinnings of the web. Pages that use GET parameters can be bookmarked, cached, and navigated to. Thus, it is undesirable for GET requests to be sent to scripts that modify data. Consequently, forms that have such effects should use the POST method, and their back-end implementation should look for POST parameters. As a happy side effect, this protects against the "request as embedded image" class of CSRF attacks.

However, there are plenty of websites on the Internet with cross-site scripting vulnerabilities, so an attacker can easily find a site from which to launch a POST-based attack. This is not to say that web applications shouldn't used HTTP POST wherever it's applicable—just that using POST is no guarantee of safety. At best, it can be considered a preliminary mitigation in a larger defense in depth strategy.

Checking the "referer" header

One solution that people often think of when presented with the problem of CSRF is to check the "referrer" header in the HTTP request. Referer checking is widely maligned in the security field because a malicious user can easily modify the referer header that their web browser sends. However, in CSRF, we are not concerned with malicious clients; the clients are the innocent victims of the attack. The referer header value should be trustworthy.

Unfortunately, we cannot depend on the fact that the referer header will accompany each request. Some web browsers never send the header, some only send the header some of the time. Additionally, proxy servers may strip out the header. In most cases, demanding that users have the right combination of software and network infrastructure so that they send the referer header usually incurs unacceptable costs. It is certainly acceptable and effective to check the referer header if it accompanies a request. Unfortunately, it is usually necessary to allows requests with a blank or missing header.

Effective CSRF solutions

Now that we've run through some common non-working solutions to CSRF vulnerabilities, we'll discuss some solutions that work. All of them are effective enough to reduce the CSRF threat to a negligible concern, but all have costs. Some are easier to implement than others, some incur heavy burdens on users, and some are more secure than others. Which one is right for you depends on your application and the circumstances of your development cycle and user base.

What is a "nonce"?

Many of these solutions involve the use of a nonce. "Nonce" is a shortened form of "cryptographic number used only once," a one-time token used in a transaction.[5] The requirements for a nonce used for CSRF protection are significantly lower than one used for cryptography. Attackers will be limited in the number of requests that they can cause their victim to send, so the nonce only needs to be somewhat difficult to predict. While there is no reason not to use a high-quality random number, such as 128 bits of cryptographically random data, or a GUID, it is acceptable to simply use a hash of two or more non-cryptographically random numbers and a static secret.

Single per-page nonce

The simplest method of CSRF protection to implement is to insert a nonce into each form and also into a special slot in the server session, and then to compare the values of these two variables when the form is submitted. Here is a pseudo-code example:

```
<%
nonce = generate_nonce()
session.nonce = nonce
%>
<form>
 <input name="field1"><br>
 <input name="field2"><br>
 <input type="submit">
 <input name="nonce" type="hidden" value="<%= nonce %>">
</form>
```

When the form is submitted, the following is executed:

```
if (post.nonce != session.nonce) {
 log_CSRF_attack()
 error_and_exit()
}
// normal form handling here
```

What this code does is verify that each request to process a request has been preceded by a request for the associated form. In other words, for each form submission, the form has actually been loaded. Since, due to the DOM security model, the attacker cannot read data from another site, the attacker cannot load the form and read the nonce value. Although this approach provides excellent protection against CSRF, it is not without problems.

The problems with this approach lie in the realm of breaking expected web behavior, rather than in security. For example, say a user is accessing a website with their browser. They then open a second window or tab in their browser and continue to browse the same site in that second window or tab.

First browser window/tab	Second browser window/tab	Server
Loads form: nonce a		Nonce a
	Loads form: nonce b	Nonce b
	Submits form	Nonce b: okay
Submits form		Nonce a: error

As can be seen, the user would normally expect both form submissions to go through, but the second one (in the "firstwindow/tab") is blocked by the server.

In addition to problems with multiple browser windows, page caching may cause similar issues with this approach.

Per-session nonce

To overcome the usability weaknesses of the per-form nonce, a per-session token can be used. In this case, a single token is created at the beginning of the session and is used throughout the session. In this pseudo-code example, the following would be in some global application file:

```
<%
function session_initiate(first_name, last_name /* etc */) {
    session.fisrt_name = first_name
    session.last_name = last_name
    /* etc */
    session.form_token = generate_form_token()
}
%>
```

Then, in the page code:

```
<%
<form>
 <input name="field1"><br>
 <input name="field2"><br>
 <input type="submit">
 <input name="form_token" type="hidden" value="<%= session.form_token %>">
</form>
```

When the form is submitted, the following is executed:

```
if (post.form_token != session.form_token) {
 log_CSRF_attack()
 error_and_exit()
}
// normal form handling here
```

The primary advantage of this method is that multiple browser windows, page caching, and other functions will not cause false positives in CSRF detection. However, it is a rather fragile solution. Since the form token has a long lifespan, it must be protected from leakage. If an attacker were to be able to recover the target's form token, they would be able to issue valid requests so long as the target's session was active.

Fortunately, the techniques that must be used to protect the token are well understood and are part of longstanding secure web development practices. The token must be secure in transport; communications should be protected via SSL, and the token should never appear in a URL, so as to prevent retrieval via a proxy or the browser history. Of course, a cross-site scripting vulnerability in the website will allow an attacker to steal the form token. However, if the attacker has located a cross-site scripting vulnerability, there's a very good chance that he or she has no interest in exploiting a CSRF vulnerability—the cross-site scripting is enough.

CAPTCHAs/confirmation

The above solutions have the significant advantage of being transparent to the user. As such, they are the preferred solutions in the vast majority of cases. There are several other more intrusive protection mechanisms. They are discussed here mainly because they may be already in place to defend against other attacks and so may give a website protection against CSRF as a bonus feature. All of these mechanisms use the core concept as the nonces described above, but they require user interaction. At their core, they have some data that the attacker cannot read (due to the same origin policy), but they require the user to enter the secret data.

CAPTCHAs are intended to prevent automated scripts from submitting forms on a website. They generally present a distorted image of some text on the web page, and require the user to retype the text into a text box. Since the attacker cannot read the CAPTCHA image, then they have no hope of determining the proper text. Note that the degree of distortion has no bearing on CSRF protection; even the worst possible CAPTCHA provides effective defense.

Some sites require users to re-enter their password or to enter a separate "confirmation password" in order to perform sensitive actions. One common example is the change password feature on most websites. Since the attacker does not know the victim's password, CSRF is not possible.

In general, any operation that requires information that cannot be predicted by an attacker who does not have access to the user's web pages or secrets will not be vulnerable to CSRF.

Protecting Yourself

While protecting websites against CSRF is relatively straightforward, it is time consuming to retrofit protection onto a site. Given that almost every site that allows users to perform operations is vulnerable to CSRF, users may want to take steps to protect themselves from CSRF attacks.

Log out

CSRF attacks generally require that the victim be logged into the target website. In order to minimize exposure to such attacks, users can limit the amount of time they spend logged into websites. Of particular concern are sites that use HTTP basic authentication. Browsers will often offer users the opportunity to save their basic authentication credentials for later use. If the user elects to do so, CSRF attacks against those websites will always work. A common example of this is the so-called "drive-by pharming"[6] class of attacks against broadband routers.

To minimize risk from CSRF attacks, users should not save basic authentication credentials, log out of websites upon completing their transactions, and avoid general web browsing while using "important" or "sensitive" websites.

Change default passwords

Many web-enabled devices ship with default user accounts, with well-known usernames and passwords. An attacker can exploit this to use CSRF to attempt to log users into the target website before performing a real CSRF attack. This is of particular concern for routers, which tend to have predictable IP addresses. Changing the default password on these devices prevents attackers from performing a CSRF login operation.

Use different browsers

Most CSRF targets require the victim to have an active session on the website in order for the attack to work. One way users can protect themselves from CSRF attacks is to use one browser for browsing sensitive, trusted sites and another for general browsing. For instance, a corporate user might use Microsoft Internet Explorer to browse her corporate intranet, and Firefox to browse the Internet. Extremely sensitive sites (such as management consoles for network infrastructure, user management consoles, financial portals, and others) might warrant a third browser.

Use a virtual machine

Using a different browser for different classes of browsing behavior is not a particularly scalable solution, considering that there are really only five major web browsers available at the moment. Furthermore, site browser requirements might limit a user's choices. One way around this problem is to use dedicated browser virtual machines (VMs), configured with a minimal OS and a web browser. While this solution has scalability problems of its own, it also has other security advantages. Many users will run a browser in a VM while visiting suspicious websites. This contains the damage in the event that the user stumbles upon some nasty malware.

Enforcement via proxies

IT administrators can enforce the use of different browsers for visiting different classes of sites by using proxy servers and configuring the proxy settings on their users' browsers. For example, access to Internet websites might only be available via a proxy server, whereas intranet sites were directly accessible. The IT administrator could configure users' Google Chrome browser to only use the proxy server (and thus only have access to Internet sites), and configure users' Internet Explorer to not use the proxy server (and thus only have access to intranet sites). This configuration would eliminate the ability for malicious sites on the Internet to launch CSRF attacks against the intranet and the organization's web-based management consoles.

Conclusion

Cross-site request forgery is a subtle attack technique that can be extremely powerful. In some cases (such as attacks against a user management system that allow an attacker to create administrative users), it can lead to the complete compromise of a web-based system. In other cases, the impact is minimal.

Website owners can defend their sites against CSRF by validating the origin of requests or by requiring secret information to accompany requests. Users may protect themselves by limiting their exposure to potential attacks.

During the three and a half years between when this paper was started and when it was finished, awareness of CSRF has greatly increased, and many libraries[7] are available to help developers protect their websites. However, the overwhelming majority of sites on the Internet remain completely vulnerable. It is my hope that this paper will help in raising awareness of the issue and the available countermeasures.

Acknowledgements

I would like to thank Brandon Eisenmann for introducing me to the concept of CSRF many years ago, his colleagues at McAfee Foundstone for reviewing this paper, and his parents for indulging his compulsion as a child to take apart every mechanical and electrical device he could get his hands on.

About the Author

Jeremiah Blatz is a security consultant at McAfee Foundstone. He is the maintainer of McAfee Foundstone's Ultimate Hacking: Web class and the service line lead for web application penetration testing. In his spare time he enjoys training in Sambo and cooking overambitious meals for his wonderful wife.

About McAfee Foundstone Professional Services

McAfee Foundstone Professional Services, a division of McAfee, offers expert services and education to help organizations continuously and measurably protect their most important assets from the most critical threats. Through a strategic approach to security, McAfee Foundstone identifies and implements the right balance of technology, people, and process to manage digital risk and leverage security investments more effectively. The company's professional services team consists of recognized security experts and authors with broad security experience with multinational corporations, the public sector, and the US military.

---

1. http://ha.ckers.org/blog/20070213/70-of-websites-under-immediate-risk-of-being-hacked/
http://www.whitehatsec.com/home/resources/files/block_0/web_app_sec_risk_report.pdf
2. http://www.gnucitizen.org/blog/csrf-ing-blogger-classic
3. http://jeremiahgrossman.blogspot.com/2006/11/browser-port-scanning-without.html
4. http://www.w3.org/MarkUp/html-spec/html-spec_8.html#SEC8.2
5. http://en.wikipedia.org/wiki/Cryptographic_nonce
6. http://en.wikipedia.org/wiki/Pharming
7. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet#Prevention_Frameworks

McAfee
2821 Mission College Boulevard
Santa Clara, CA 95054
888 847 8766
www.mcafee.com

36006wp_csrf-attack-defense_1011_fnl_ETMG