Cross-Site Request Forgery(CSRF)

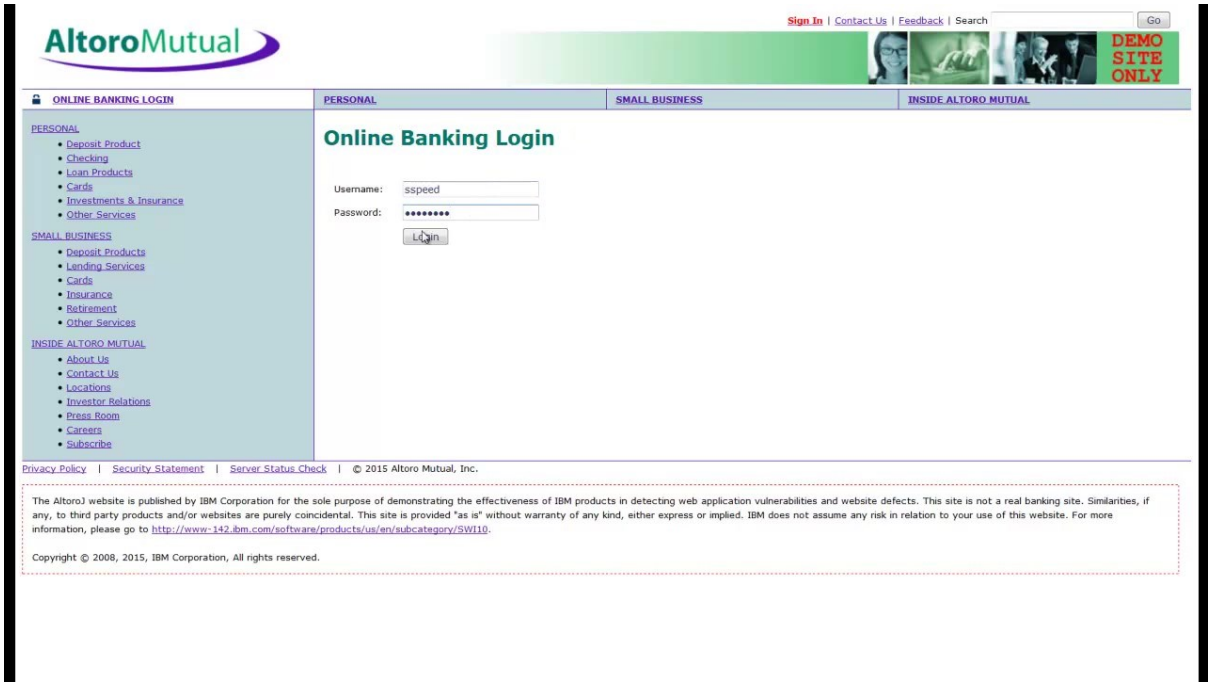
This type of security vulnerability was among the OWASP top 10 category, and is still, a frequently exploited vulnerability type.

What is a CSRF vulnerability?

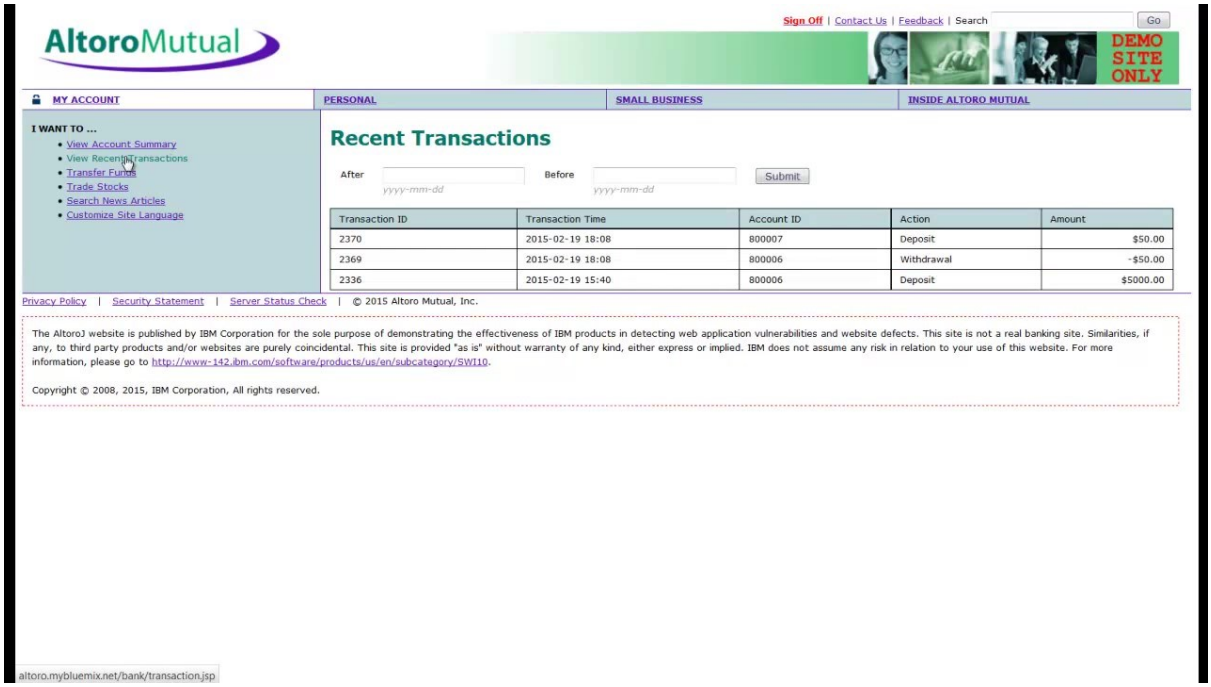
CSRF is a web application vulnerability that makes it possible for an attacker to force a user to unknowingly perform actions while they are logged into an application. Attackers commonly use CSRF to target sites that contain sensitive information like cloud storage, social media, banking and on-line shopping because of the user information and actions available in these applications.

Depending on the action being performed, a CSRF vulnerability can have serious consequences for the user using the web application. Users are usually unaware that malicious actions are being performed. So, this attack is quite a stealth attack. The attacker tricks the user into clicking a seemingly innocent URL, which while loading tricks the browser into visiting a link to fetch an HTML element. If the User is logged in, the action will be executed due to cookie-authorisation. For e.g. a bank transaction, or a social-media post, etc.

Demonstration



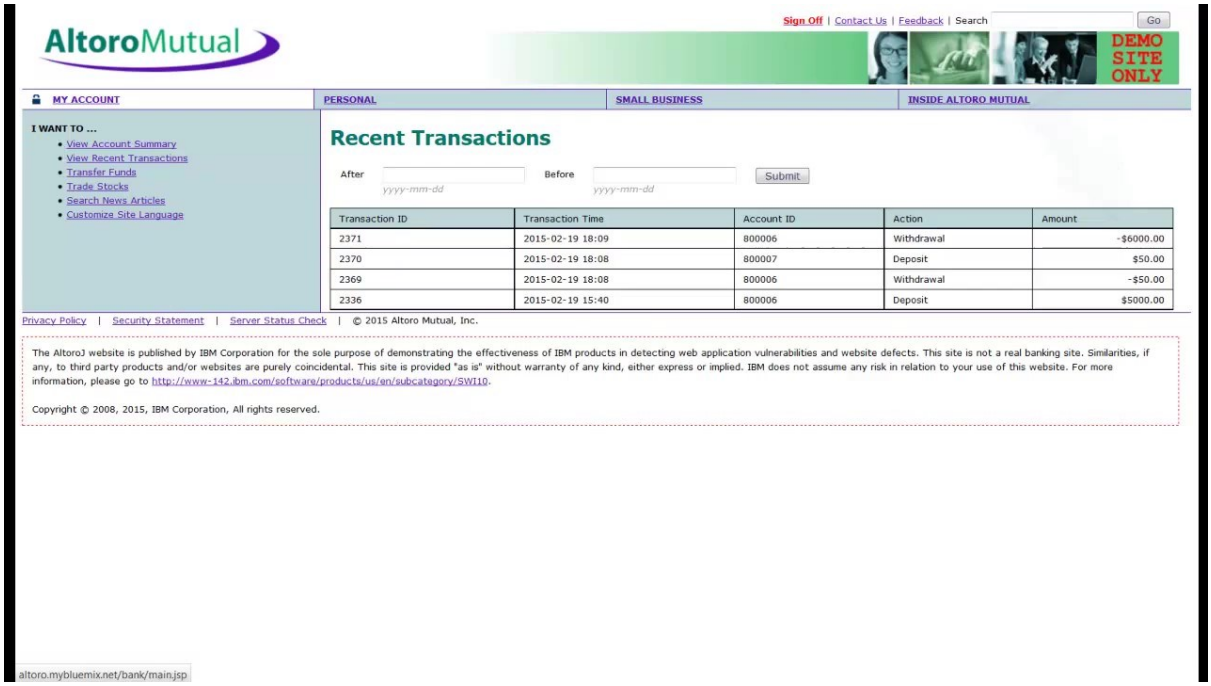
The user logs in with his credentials.



This is a list of the user's recent transactions.

Click Here!!!

The user clicks on a malicious link like this. This user is convinced in many ways to do this by social engineering. Clicking the link executes the transaction in the user's browser.



An amount of $6000 is transferred into the attacker's account! Without the user knowing.

How to avoid CSRF?

First, what doesn't work:

Using a secret cookie

Remember that all cookies, even the *secret* ones, will be submitted with every request. All authentication tokens will be submitted regardless of whether or not the end-user was tricked into submitting the request. Furthermore, session identifiers are simply used by the application container to associate the request with a specific session object. The session identifier does not verify that the end-user intended to submit the request.

Only accepting POST requests

Applications can be developed to only accept POST requests for the execution of business logic. The misconception is that since the attacker cannot construct a malicious link, a CSRF attack cannot be executed. Unfortunately, this logic is incorrect. There are numerous methods in which an attacker can trick a victim into submitting a forged POST request, such as a simple form hosted in an attacker's Website with hidden values. This form can be triggered automatically by JavaScript or can be triggered by the victim who thinks the form will do something else.

A number of flawed ideas for defending against CSRF attacks have been developed over time. Here are a few that OWASP recommends you avoid.

Multi-Step Transactions

Multi-Step transactions are not an adequate prevention of CSRF. As long as an attacker can predict or deduce each step of the completed transaction, then CSRF is possible.

URL Rewriting

This might be seen as a useful CSRF prevention technique as the attacker cannot guess the victim's session ID. However, the user's session ID is exposed in the URL.

HTTPS

HTTPS by itself does nothing to defend against CSRF.

However, HTTPS should be considered a prerequisite for any preventative measures to be trustworthy.

How to prevent CSRF?[1]

[1] This section has been adapted from OWASP prevention cheat sheet, find the reference here

OWASP recommends two separate checks as your standard CSRF defense that does not require user intervention. This discussion ignores for the moment deliberately allowed cross origin requests (e.g., CORS). Your defenses will have to adjust for that if that is allowed.

1. Check standard headers to verify the request is same origin
2. AND Check CSRF token

Each of these is discussed next.

Verifying Same Origin with Standard Headers

There are two steps to this check:

1. Determining the origin the request is coming from (source origin)
2. Determining the origin the request is going to (target origin)

Both of these steps rely on examining an HTTP request header value. Although it is usually trivial to spoof any header from a browser using JavaScript, it is generally impossible to do so in the victim's browser during a CSRF attack, except via an XSS vulnerability in the site being attacked with CSRF. More importantly for this recommended Same Origin check, a number of HTTP request headers can't be set by JavaScript because they are on the 'forbidden' headers list. Only the browsers themselves can set values for these headers, making them more trustworthy because not even an XSS vulnerability can be used to modify them.

The Source Origin check recommended here relies on three of these protected headers: Origin, Referer, and Host, making it a pretty strong CSRF defense all on its own.

Identifying Source Origin

To identify the source origin, we recommend using one of these two standard headers that almost all requests include one or both of:

- Origin Header
- Referer Header

Checking the Origin Header

If the Origin header is present, verify its value matches the target origin. The Origin HTTP Header standard was introduced as a method of defending against CSRF and other Cross-Domain attacks. Unlike the Referer, the Origin header will be present in HTTP requests that originate from an HTTPS URL. If the Origin header is present, then it should be checked to make sure it matches the target origin.

This defense technique is specifically proposed in section 5.0 of Robust Defenses for Cross-Site Request Forgery. This paper proposes the creation of the Origin header and its use as a CSRF defense mechanism.

Following a 302 redirect cross-origin: In this situation, the Origin is not included in the redirected request because that may be considered sensitive information you don't want to send to the other origin. But since we recommend rejecting requests that don't have both Origin and Referer headers, this is OK, because the reason the Origin header isn't there is because it is a cross-origin redirect.

Checking the Referer Header

If the Origin header is not present, verify the hostname in the Referer header matches the target origin. Checking the Referer is a commonly used method of preventing CSRF on embedded network devices because it does not require any per-user state. This makes Referer a useful method of CSRF prevention when memory is scarce or server-side state doesn't exist. This method of CSRF mitigation is also commonly used with unauthenticated requests, such as requests made prior to establishing a session state which is required to keep track of a synchronization token.

In both cases, just make sure the target origin check is strong. For example, if your site is "site.com" make sure "site.com.attacker.com" doesn't pass your origin check (i.e., match through the trailing / after the origin to make sure you are matching against the entire origin).

What to do when Both Origin and Referer Headers Aren't Present

If neither of these headers is present, which should be VERY rare, you can either accept or block the request. **OWASP recommend blocking**, particularly if you aren't using a random CSRF token as your second check. You might want to log when this happens for a while and if you basically never see it, start blocking such requests.

Identifying the Target Origin

You might think its easy to determine the target origin, but its frequently not. The first thought is to simply grab the target origin (i.e., its hostname and port #) from the URL in the request. However, the application server is frequently sitting behind one or more proxies and the original URL is different from the URL the app server actually receives. If your application server is directly accessed by its users, then using the origin in the URL is fine and you're all set.

Determining the Target Origin When Behind a Proxy

If you are behind a proxy, there are a number of options to consider:

1. Configure your application to simply know its target origin
2. Use the Host header value
3. Use the X-Forwarded-Host header value

Its your application, so clearly you can figure out its target origin and set that value in some server configuration entry. This would be the most secure approach as its defined server side so is a trusted value. However, this can be problematic to maintain if your application is deployed in many different places, e.g., dev, test, QA, production, and possibly multiple production instances. Setting the correct value for each of these situations can be difficult, but if you can do it, that's great.

If you would prefer the application figure it out on its own, so it doesn't have to be configured differently for each deployed instance, OWASP recommends using the Host family of headers. The Host header's purpose is to contain the target origin of the request. But, if your app server is sitting behind a proxy, the Host header value is most likely changed by the proxy to the target origin of the URL behind the proxy, which is different than the original URL. This modified Host header origin won't match the source origin in the original Origin or Referer headers.

However, there is another header called X-Forwarded-Host, whose purpose is to contain the original Host header value the proxy received. Most proxies will pass along the original Host header value in the X-Forwarded-Host header. So that header value is likely to be the target origin value you need to compare to the source origin in the Origin or Referer header.

Verifying the Two Origins Match

Once you've identified the source origin (from either the Origin or Referer header), and you've determined the target origin, however you choose to do so, then you can simply compare the two values and if they don't match you know you have a cross-origin request.

CSRF Specific Defense

Once you have verified that the request appears to be a same origin request so far, second check is recommended as an additional precaution to really make sure. This second check can involve custom defense mechanisms using CSRF specific tokens created and verified by your application or can rely on the presence of other HTTP headers depending on the level of rigor/security you want.

There are numerous ways you can specifically defend against CSRF. We recommend using one of the following (in ADDITION to the check recommended above):

1. Synchronizer (i.e.,CSRF) Tokens (requires session state)

Approaches that do require no server side state:

2. Double Cookie Defense
3. Encrypted Token Pattern
4. Custom Header - e.g., X-Requested-With: XMLHttpRequest

These are listed in order of strength of defense. So use the strongest defense that makes sense in your situation.

Personal Safety CSRF Tips for Users

Since CSRF vulnerabilities are reportedly widespread, it is recommended to follow the following best practices to mitigate risk. These include:

- Logoff immediately after using a Web application
- Do not allow your browser to save username/passwords, and do not allow sites to "remember" your login
- Do not use the same browser to access sensitive applications and to surf the Internet freely (tabbed browsing).
- The use of plugins such as No-Script makes POST based CSRF vulnerabilities difficult to exploit. This is because JavaScript is used to automatically submit the form when the exploit is loaded. Without JavaScript the attacker would have to trick the user into submitting the form manually.

Integrated HTML-enabled mail/browser and newsreader/browser environments pose additional risks since simply viewing a mail message or a news message might lead to the execution of an attack.

References and Whitepapers

1. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

2. https://www.owasp.org/index.php/Reviewing_code_for_Cross-Site_Request_Forgery_issues

3. https://www.owasp.org/index.php/Testing_for_CSRF_(OTG-SESS-005)

4. https://www.ibm.com/developerworks/library/se-appscan-detect-csrf-xsrf/index.html

5. https://en.wikipedia.org/wiki/Cross-site_request_forgery