acunetix.com

CSRF Attacks: Anatomy, Prevention, and XSRF Tokens | Acunetix

11-14 minutes

Cross-site Request Forgery, also known as CSRF, Sea Surf, or XSRF, is an attack whereby an attacker tricks a victim into performing actions on their behalf. The impact of the attack depends on the level of permissions that the victim has. Such attacks take advantage of the fact that a website completely trusts a user once it can confirm that the user is indeed who they say they are.

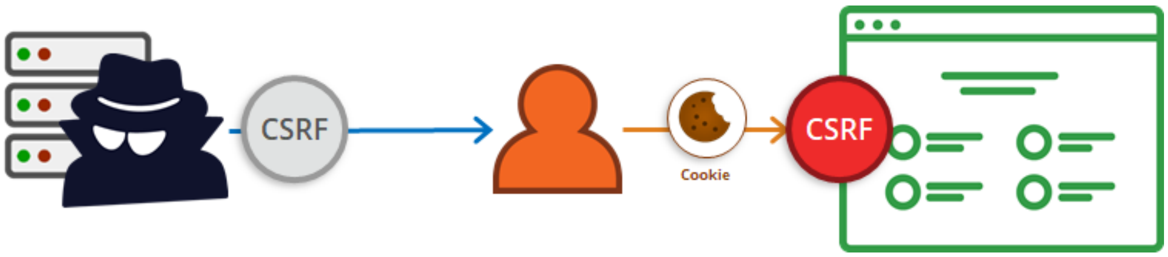
Cross-site Request Forgery is considered a sleeping giant in the world of web application security. It is often not taken as seriously as it should even though it can prove to be a stealthy and powerful attack if executed properly. It is also a common attack, which is why it has secured a spot on the OWASP Top 10 list several times in a row. However, an exploited Cross-site Scripting vulnerability (XSS) is more of a risk than any CSRF vulnerability because CSRF attacks have a major limitation. CSRF only allows for state changes to occur and therefore the attacker cannot receive the contents of the HTTP response.

How Are CSRF Attacks Executed

There are two main parts to executing a Cross-site Request Forgery attack. The first one is tricking the victim into clicking a link or loading a page. This is normally done through social engineering and malicious links. The second part is sending a crafted, legitimate-looking request from the victim's browser to the website. The request is sent with values chosen by the attacker including any cookies that the victim has associated with that website. This way, the website knows that this victim can perform certain actions on the website. Any request sent with these HTTP credentials or cookies will be considered legitimate, even though the victim would be sending the request on the attacker's command.

When a request is made to a website, the victim's browser checks if it has any cookies that are associated with the origin of that website and that need to be sent with the HTTP request. If so, these cookies are included in all requests sent to this website. The cookie value typically contains authentication data and such cookies represent the user's session. This is done to provide the user with a seamless experience, so they are not required to authenticate again for every page that they visit. If the website approves of the session cookie and considers the user session still valid, an attacker may use CSRF to send requests as if the victim was sending them. The website is unable to distinguish between requests being sent by the attacker and those sent by the victim since requests are always being sent from the victim's browser with their own cookie. A CSRF attack simply takes advantage of the fact that the browser sends the cookie to the website automatically with each request.

Cross-site Request Forgery will only be effective if a victim is authenticated. This means that the victim must be logged in for the attack to succeed. Since CSRF attacks are used to bypass the authentication process, there may be some elements that are not affected by these attacks even though they are not protected against them, such as publicly accessible content. For example, a public contact form on a website is safe from CSRF. Such HTML forms do not require the victim to have any privileges for form submission. CSRF only applies to situations where a victim is able to perform actions that are not accessible to everyone.



A CSRF Attack Example Using a GET Request

HTTP GET is by its very nature meant to be an idempotent request method. This means that this HTTP method should not be used to perform state changes. Sending a GET request should never cause any data to change. However, some web apps still use GET instead of the more appropriate POST to perform state changes for operations such as changing a password or adding a user.

When the victim clicks the link provided by the attacker using social engineering, the victim is directed to the attacker's malicious site. This website executes a script that triggers the user's web browser to send an unsolicited request. The victim is not aware that this unsolicited client-side request is being sent.

However, server-side it appears as if the user sent the request because it includes cookies used to verify that the user is who they say they are.

Let's imagine that www.example.com processes fund transfers using a GET request that includes two parameters: the amount that is to be transferred and the identifier of the person to receive the money transfer. The below example shows a legitimate URL, which will request that the web app transfers 100,000 units of the appropriate currency to Fred's account.

```
http://example.com/transfer?amount=1000000&
account=Fred
```

The request includes a cookie that represents the authenticated user so there is no need to define the source account for the transfer. If a normal user accesses this URL, they need to authenticate so that the application knows the account from which funds are to be withdrawn. Using CSRF, we can trick a victim into sending the request that the attacker wants while authenticated as the victim.

If the exploited application expects a GET request, the attacker can include a malicious `<img>` tag on their own website. Instead of linking to an image, this tag sends a request to the bank's web app:

```
<img data-fr-src="http://example.com
/transfer?amount=1000000&account=Fred" />
```

Under normal circumstances, the user's browser automatically sends cookies that are related to that website. This causes the victim to perform a state change on behalf of the attacker. In this case, the state change is a transfer of funds.

Note that this example is very simple and it does not necessarily reflect the real-world but it shows very well how CSRF attacks work. However, similar vulnerabilities based on GET appeared in popular software in the past (read more about it on Wikipedia).

CSRF Attacks Using POST Requests

Most state-changing requests are done using HTTP POST requests. This means that web apps are more likely to accept POST instead of GET when a state change is involved. In the case of POST, the user's browser sends parameters and values in the request body and not the URL as in the case of a GET request.

Tricking a victim into sending a POST request may be slightly more difficult. With a GET request, the attacker only needs the victim to send a URL with all the necessary information. In the case of POST, a request body must be appended to the request. However, an attacker can design a malicious website to include JavaScript that causes the user's browser to send an unsolicited POST request as soon as the page loads.

The following JavaScript example shows the onload function, which automatically sends a request from the victim's browser as soon as the page loads.

```
<body onload="document.csrf.submit()">

<form action="http://example.com/transfer"
method="POST" name="csrf">
```

```
        <input type="hidden" name="amount"
value="1000000">
        <input type="hidden" name="account"
value="Fred">
</form>
```

As soon as the page loads, the JavaScript onload function ensures that the hidden form is submitted, which will in turn send the POST request. The form includes two parameters and their values that have been set up by the attacker. The POST target, example.com, identifies the request as legitimate because it includes the victim's cookies.

An attacker can also make use of an IFrame with attributes that make it invisible. Using the same onload function, the attacker can load the IFrame containing a malicious web page and cause a request to be sent as soon as the IFrame loads. Another option is to use XMLHttpRequest technology.

Preventing CSRF Vulnerabilities

Security experts propose many CSRF prevention mechanisms. This includes, for example, using a referer header, using the `HttpOnly` flag, sending an `X-Requested-With` custom header using jQuery, and more. Unfortunately, not all of them are effective in all scenarios. In some cases, they are ineffective and in other cases, they are difficult to implement in a particular application or have side effects. The following implementations prove to be effective for a variety of web apps while still providing protection against CSRF attacks. For more advanced CSRF prevention options, see the CSRF prevention cheat sheet managed by OWASP.

What Are CSRF Tokens

The most popular method to prevent Cross-site Request Forgery is to use a challenge token that is associated with a particular user and that is sent as a hidden value in every state-changing form in the web app. This token, called an *anti-CSRF token* (often abbreviated as *CSRF token*) or a *synchronizer token*, works as follows:

- The web server generates a token and stores it
- The token is statically set as a hidden field of the form
- The form is submitted by the user
- The token is included in the POST request data
- The application compares the token generated and stored by the application with the token sent in the request
- If these tokens match, the request is valid
- If these tokens do not match, the request is invalid and is rejected

This CSRF protection method is called the *synchronizer token pattern*. It protects the form against Cross-site Request Forgery attacks because an attacker would also need to guess the token to successfully trick a victim into sending a valid request. The token should also be invalidated after some time and after the user logs out. Anti-CSRF tokens are often exposed via AJAX: sent as headers or request parameters with AJAX requests.

For an anti-CSRF mechanism to be effective, it needs to be cryptographically secure. The token cannot be easily guessed, so it cannot be generated based on a predictable pattern. We also recommend to use anti-CSRF options in popular frameworks such as AngularJS and refrain from creating own mechanisms, if possible. This lets you avoid errors and makes the implementation quicker and easier.

Same-Site Cookies

CSRF attacks are only possible because cookies are always sent with any requests that are sent to a particular origin related to that cookie (see the definition of the same-origin policy). You can set a flag for a cookie that turns it into a same-site cookie. A same-site cookie is a cookie that can only be sent if the request is being made from the origin related to the cookie (not cross-domain). The cookie and the request source are considered to have the same origin if the protocol, port (if applicable) and host (but not the IP address) are the same for both.

A current limitation of same-site cookies is that unlike for example Chrome or Firefox, not all current browsers support them and older browsers do not work with web apps that use same-site cookies (click here for a list of supported browsers). At the moment, same-site cookies are better suited as an additional defense layer due to this limitation. Therefore, you should only use them along with other CSRF protection mechanisms.

Conclusion

Cookies are intrinsically vulnerable to CSRF because they are automatically sent with each request. This allows attackers to easily craft malicious requests that lead to CSRF. Although the attacker cannot obtain the response body or the cookie itself, they can perform actions with the victim's elevated rights. The impact of a CSRF vulnerability is related to the privileges of the victim. While sensitive information retrieval is not the main scope of a CSRF attack, state changes may have an adverse effect on the exploited web application.

Fortunately, it's easy to test if your website or web application is vulnerable to CSRF and other vulnerabilities by running an automated web scan using the Acunetix vulnerability scanner, which includes a specialized CSRF scanner module. Take a demo and find out more about running CSRF scans against your website or web application.