Learning Objectives: CSRF

Learners will be able to...

- **Describe what CSRF (Cross-Site Request Forgery) is**
- **Explain the difference between CSRF and XSS (Cross-Site Scripting)**
- **Suggest two methods to prevent a CSRF attack**

info

Make Sure You Know

- HTML basics
- JavaScript basics

What Is CSRF?

Cross-Site Request Forgery

Also known as session riding or the one-click attack, a **cross-site request forgery (CSRF)** is a web application cyberattack that tricks victims into unknowingly performing actions on the attacker's behalf. CSRF attacks exploit a security flaw in web applications where they cannot differentiate between a dangerous and legitimate request within an authenticated user session.

These perpetrators typically launch CSRF attacks using social engineering techniques to trick the victim into loading a page or clicking a link containing a malicious request. The link sends a malicious request from the authenticated user's browser to the target website.

For most websites, browser requests inherently include session information such as session cookies, a valid token, or login credentials that the website associates with the user. If the authenticated user is already in an active session with the target website, the site treats the new malicious request as a valid request from the user and executes it.

CSRF Attack Types

A web application is vulnerable to Cross-Site Request Forgery if it fails to differentiate between an illegitimate and valid request. The attack is only successful if the authenticated user is in an active session with the web application.

Login CSRF Attack

An attacker can use CSRF to obtain the victim's private data via a special form of the attack, known as **login CSRF**. The attacker forces a non-authenticated user to log in to an account the attacker controls. If the victim does not realize this, they may unknowingly share personal data such as credit card information to the account. The attacker can then log into the user's actual account to view this data, along with the victim's activity history on the legitimate web application.

Stored CSRF Attack

It's sometimes possible to store the CSRF attack on the vulnerable site itself. Such vulnerabilities are called **stored CSRF flaws**. This can be accomplished by simply storing an `IMG` or `IFRAME` tag in a field that accepts HTML, or by a more complex cross-site scripting (XSS) attack. If the site contains both CSRF and XSS, it is much more dangerous. In particular, the likelihood of the victim's personal information being stolen is increased because they are more likely to trigger the attack. Because the web application itself is legitimate, the user is much more likely to have their guard down and may be more willing to share information that the attacker wants.

Summing up, **Cross-Site Request Forgery (CSRF)** is an attack that tricks an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions that the attacker wants. If the victim has a regular or consumer account, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim has an administrative account, CSRF can compromise the entire web application.

How Does the Attack Work?

For a CSRF attack to be possible, three key conditions must be in place:

1. **A relevant action** - There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action that can alter user-specific data (such as changing the user's own password).
2. **Cookie-based session handling** - Performing the action involves issuing one or more HTTP requests, and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests.
3. **No unpredictable request parameters** - The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the attempt will be unsuccessful if the attacker needs to know the value of the existing password.

Here is an example. Suppose an application contains a function that lets the user change the email address on their account. When a user performs this action, they make an HTTP request like the following:

```
POST /change_email HTTP/1.1
Host: somewebsite.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=somesession

email=username@hostname.com
```

This meets the conditions required for CSRF:
* The action of changing the email address on a user's account is of interest to an attacker. Following this action, the attacker will typically be able to trigger a password reset and take full control of the user's account.
* The application uses a session cookie to identify which user issued the request. There are no other tokens or mechanisms in place to track user sessions.
* The attacker can easily determine the values of the request parameters that are needed to perform the action.

With these conditions in place, the attacker can construct a web page containing the following HTML:

```
<html>
    <body>
        <form action="https://somewebsite.com/change_email"
        method="POST">
            <input type="hidden" name="email"
        value="hacker@hostname.net" />
        </form>
        <script>
            document.forms[0].submit();
        </script>
    </body>
</html>
```

If a victim user visits the attacker's web page, the following will happen:

- The attacker's page will trigger an HTTP request to the vulnerable web site.
- If the user is logged in to the vulnerable web site, their browser will automatically include their session cookie in the request (assuming `SameSite` cookies are not being used).
- The vulnerable web site will process the request normally, treat it as having been made by the victim user, and change their email address.

An alternative method an attacker might perform is to use `GET` requests which could be processed more simply by sending a link to a victim user or a zero-based image in an email.

```
<a href="https://somewebsite.com/change_email?
        email=hacker@hostname.net">
  View my Pictures!
</a>


<img
  src="https://somewebsite.com/change_email?
        email=hacker@hostname.net"
  width="0"
  height="0"
  border="0"
>
```

CSRF vs. XSS

Both CSRF and XSS are client-side attacks that abuse the same-origin policy and exploit the trusting relationship between the web application and an unsuspecting user.

There are, however, a few fundamental differences between XSS and CSRF attacks which includes:

- XSS attacks follow a two-way attack pattern, which allows the attacker to execute a malicious script, access the response, and send follow-up sensitive data to a destination of the attacker's choice. On the other hand, CSRF is a one-way attack mechanism, meaning the attacker can only initiate the HTTP request but not retrieve the response in return for the initiated request.
- CSRF attacks require the authenticated user to be in an active session while XSS attacks do not. In an XSS attack, payloads (packets of transmitted data) can be stored and delivered whenever the user logs in.
- CSRF attacks have a limited scope which requires the user to perform certain actions, such as clicking a malicious link or visiting the hacker's website. On the contrary, XSS attacks occur through the execution of malicious scripts to perform any activity the attacker wants, thus widening the scope of the attack.
- In XSS attacks, the malicious code is stored within the site, whereas in CSRF attacks, the malicious code is stored within third-party sites that the victim user is tricked into accessing.

CSRF Prevention Mechanisms

Here are some suggested ways to prevent CSRF attacks:

- Using a synchronizer token pattern
- Using double-submit cookies
- Using HTTP standard headers to validate the origin of requests
- UI-based verification such as CAPTCHA-based authorization and MFA
- Using `SameSite` Cookies for request origin management
- Use of CSRF tokens

For additional information, see the PDF on the left which is a “cheat sheet” offered by OWASP (Open Web Application Security Project), a nonprofit foundation that works to improve the security of software.

Use Built-In Or Existing CSRF Implementations for CSRF Protection

Synchronizer token defenses have been built into many frameworks. It is strongly recommended that you research if the framework you are using has an option to achieve CSRF protection by default before trying to build your custom token generating system. For example, .NET has built-in protection that adds a token to CSRF vulnerable resources. You are responsible for properly configuring your management systems (such as key management and token management) before using these built-in CSRF protections that generate tokens to guard CSRF vulnerable resources.

Synchronizer Token Pattern

CSRF tokens should be generated on the server-side. They can be generated once per user session or for each request. Per-request tokens are more secure than per-session tokens as the time range for an attacker to exploit the stolen tokens is minimal. However, this may result in usability concerns. For example, the “Back” button browser capability is often hindered as the previous page may contain a token that is no longer valid. Interaction with this previous page will result in a CSRF false positive security event on the server. If per-session token is implemented after the initial generation of the token, the value is stored in the session and is used for each subsequent request until the session expires.

When a request is issued by the client, the server-side component must verify the existence and validity of the token in the request compared to the token found in the user session. If the token was not found within the request, or the value provided does not match the value within the user session, then the request should be aborted. Additional actions such as logging the event as a potential CSRF attack in progress should also be considered.

CSRF tokens should be:

- Unique per user session
- Secret
- Unpredictable (large random value generated by a secure method)

CSRF tokens prevent CSRF attacks because without a token, an attacker cannot create a valid request to the backend server.

For the Synchronised Token Pattern, CSRF tokens should not be transmitted using cookies.

The CSRF token can be transmitted to the client as part of a response payload, such as an HTML or JSON response. It can then be transmitted back to the server as a hidden field on a form submission, or via an AJAX request as a custom header value or part of a JSON payload. Make sure that the token is not leaked in the server logs, or in the URL. CSRF tokens in `GET` requests are potentially leaked at several locations, such as the browser history, log files, network appliances that log the first line of an HTTP request, and Referer headers if the protected site links to an external site.

For example:

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"
value="LONG_AND_SECURE_TOKEN">
[...]
</form>
```

Inserting the CSRF token in the custom HTTP request header via JavaScript is considered more secure than adding the token in the hidden field form parameter because it uses custom request headers.

Double Submit Cookie

If maintaining the state for a CSRF token at the server side is problematic, an alternative defense is to use the **double submit cookie** technique. This technique is easy to implement and is stateless. In this technique, we send a random value as both a cookie and as a request parameter which the server will verify if the cookie value matches the request value. When a user visits the site (even before authenticating to prevent login CSRF), the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine separate from the session identifier. The site then requires that every transaction request include this pseudorandom value as a hidden form value (or other request parameter/header). If both of them match at the server side, the server accepts it as a legitimate request and if they don't match, it will reject the request.

To enhance the security of this method, include the token in an encrypted cookie, other than the authentication cookie (since they are often shared within subdomains), and then have the server side match it (after decrypting the encrypted cookie) with the token in the hidden form field or parameter/header for AJAX calls. This works because a sub domain has no way to over-write a properly crafted encrypted cookie without the necessary information such as encryption key.

A simpler alternative to an encrypted cookie is to use HMAC (Hash-based Message Authentication Code) - a specific type of message authentication code involving hash functions and secret keys. Hash the token with a secret key known only by the server side and place this value in a cookie. This is similar to an encrypted cookie (both require knowledge only the server side holds), but is less computationally intensive than encrypting and decrypting a cookie. Whether cookie encryption or HMAC is used, an attacker won't be able to recreate the cookie value from the plain token without knowledge of the server side's secret keys or values.

SameSite Cookie Attribute

SameSite is a cookie attribute (similar to HTTPOnly, Secure, etc.) which aims to mitigate CSRF attacks. It is defined in RFC6265bis (click on the link to learn more). This attribute helps the browser decide whether to send cookies along with cross-site requests or not. Possible values for this attribute are `Lax`, `Strict`, or `None`.

The `Strict` value will prevent the cookie from being sent by the browser to the target site in all cross-site browsing contexts, even when following a regular link. For example, for a GitHub-like website this would mean that if a logged-in user follows a link to a private GitHub project posted on a corporate discussion forum or email, GitHub will not receive the session cookie and the user will not be able to access the project. A bank website however doesn't want to allow any transactional pages to be linked from external sites, so the `Strict` flag would be most appropriate.

On the other hand, the default `Lax` value provides a reasonable balance between security and usability for websites that want to maintain a user's logged-in session after the user arrives from an external link. In the above GitHub scenario, the session cookie would be allowed when following a regular link from an external website but it will be blocked when a CSRF-prone request method is detected such as POST. The only cross-site requests that are allowed in `Lax` mode are the ones that have top-level navigations and are also safe HTTP methods.

Lastly, the `None` value will enable cookies to be sent in all contexts (in responses to both first-party and cross-site requests).

Again, for more details on SameSite values, click this link.

Here are examples of cookies using some of these attributes:

```
Set-Cookie: JSESSIONID=xxxxx; SameSite=Strict
Set-Cookie: JSESSIONID=xxxxx; SameSite=Lax
```

All desktop browsers and almost all mobile browsers now support the SameSite attribute. Note that Chrome has announced that they will mark cookies as `SameSite=Lax` by default starting from Chrome 80 (released in February 2020). Firefox and Edge are both planning to follow suit. Additionally, the `Secure` flag will be required for cookies that are marked as `SameSite=None`.

It is important to note that this attribute should be implemented as an additional layer of defense. By itself, it is not secure enough. This attribute protects the user as long as their browser supports its implementation. However, it should should not replace something like a CSRF Token, which can offer much more security. The best strategy is to implement multiple techniques that can co-exist in order to protect the user in a more robust way.

Verifying Origin With Standard Headers

Another method of mitigating a CSRF attack is verifying origin with standard headers. There are two steps to this process, both of which rely on examining an HTTP request header value.

First, determine the origin the request is coming from (**source origin**). This can be cone via the Origin or Referer headers. Second, determine the origin the request is going to (**target origin**). Then, at the server side, verify if both requests match. If they do, we accept the request as legitimate (meaning it's the same origin request). If they don't, we discard the request (meaning that the request originated from a cross-domain). Reliability on these headers comes from the fact that they cannot be altered programmatically since they fall under a *forbidden* headers list, meaning that only the browser can set them.

Identifying Source Origin (via Origin/Referer header)

CHECK THE ORIGIN HEADER

If the Origin header is present, verify that its value matches the target origin. Unlike the Referer, the Origin header will be present in HTTP requests that originate from an HTTPS URL.

CHECK THE REFERER HEADER

If the Origin header is not present, verify that the hostname in the Referer header matches the target origin. 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, make sure the target origin check is strong. For example, if your site is `example.org` make sure `example.org.attacker.com` does not pass your origin check. In other words, be sure to match through the entire origin, not just a part of it.

If neither of these headers are present, you can either accept or block the request. We recommend blocking. Alternatively, you might want to log all such instances first, monitor their use cases/behavior for a period of time,

and then start blocking the requests after you are confident that they are not legitimate.

Identifying the Target Origin

You might think it's easy to determine the target origin, but it's frequently not. The first thought might be to simply grab the target origin (e.g. its hostname and port #) from the URL in the request. However, the application server is frequently sitting behind one or more proxies so 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 would be all set.

If your application server is behind a proxy, however, there are a number of options to consider.

- Configure your application to always know its target origin. It's your application, so you can find its target origin and set that value in some server configuration entry. This would be the most secure approach as it will be defined on the server side, so it is a more trusted value. However, this might be problematic to maintain if your application is deployed in many environments (e.g. dev, test, QA, production, and possibly multiple other instances). Setting the correct value for each of these situations might be difficult, but if you can do it via some central configuration that will enable your instances to retrieve this value, then you will be good hands. (**Note:** Make sure this central configuration is maintained securely because a major part of your CSRF defense depends on it.)

- Use the Host header value. If you prefer that the application find its own target so it doesn't have to be configured for each deployed instance, we recommend 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.

- Use the X-Forwarded-Host header value. To avoid the issue of a proxy altering the host header, there is another header called the 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 going to be the target origin value you need to compare to the source origin in the Origin or Referer header.