21. Cross-Site Request Forgery (CSRF)

21. Cross-Site Request Forgery (CSRF)

21.1. CSRF Attacks

Using cookies and session tokens to keep a user logged in has some associated security risks. In a cross-site request forgery (CSRF) attack, the attacker forces the victim to make an unintended request. The victim's browser will automatically attach the session token cookie to the unintended request, and the server will accept the request as coming from the victim.

For example, suppose a website has an endpoint `http://example.com/logout`. To log out, a user makes a GET request to this URL with the appropriate session token attached, and the server checks the session token and performs the logout. If an attacker can trick a victim into clicking this link, the victim will be logged out of the website without their knowledge.

CSRF attacks can also be executed on URLs with more malicious actions. For example, a GET request to `https://bank.com/transfer?amount=100&recipient=mallory` with a valid session token might send $100 to Mallory. An attacker could send an email to the victim with the following HTML snippet:

<img src="https://bank.com/transfer?amount=100&recipient=mallory" />

This will cause the browser to try and fetch an image from the malicious URL by making a GET request. Because the browser automatically attaches the session token to the request, this causes the victim to unknowingly send $100 to Mallory.

It is usually bad practice to have HTTP GET endpoints that can change server state, so this type of CSRF attack is less common in practice. However, CSRF attacks are still possible over HTTP POST requests. HTML forms are a common example of a web feature that generates HTTP POST requests. The user fills in the form fields, and when they click the Submit button, the browser generates a POST request with the filled-out form fields. Consider the following HTML snippet on an attacker's webpage:

```
<form name=evilform action=https://bank.com/transfer>
  <input name=amount value=100>
  <input name=recipient value=mallory>
</form>
<script>document.evilform.submit();</script>
```

When the victim visits the attacker's website, this HTML snippet will cause the victim's browser to make a POST request to `https://bank.com/transfer` with form input values that transfer $100 to Mallory. Like before, the victim's browser automatically attaches the session token to the request, so the server accepts this POST request as if it was from the victim.

21.2. Defense: CSRF Token

A good defense against CSRF attacks is to include a CSRF token on webpages. When a legitimate user loads a webpage from the server with a form, the server will randomly generate a CSRF token and include it as an extra field in the form. (In practice, this field often has a hidden attribute set so that it's only visible in the HTML, so users don't see random strings every time they submit a form.) When the user submits the form, the form will include the CSRF token, and the server will check that the CSRF token is valid. If the CSRF token is invalid or missing, the server will reject the request.

To implement CSRF tokens, the server needs to generate a new CSRF token every time a user requests a form. CSRF tokens should be random and unpredictable so an attacker cannot guess the CSRF token. The server also needs to maintain a mapping of CSRF tokens to session tokens, so it can validate that a request with a session token has the correct corresponding CSRF token. This may require the server to store a large amount of state if it expects heavy traffic.

If an attacker tries the attack in the previous section, the malicious form they create on their website will no longer contain a valid CSRF token. The attacker could try querying the server for a CSRF token, but it would not properly map to the victim's session token, because the victim never requested the form legitimately.

21.3. Defense: Referer Validation

Another way to defend against CSRF tokens is to check the Referer[1] field in the HTTP header. When a browser issues an HTTP request, it includes a Referer header which indicates which URL the request was made from. For example, if a user fills out a form from a legitimate bank website, the Referer header will be set to `bank.com`, but if the user visits the attacker's website and the attacker fills out a form and submits it, the Referer header will be set to `evil.com`. The server can check the Referer header on each request and reject any requests that have untrusted or suspicious Referer headers.

Referer validation is a good defense if it is included on every request, but it poses some problems if someone submits a request with the Referer header left blank. If a server accepts requests with blank Referer headers, it may be vulnerable to CSRF attacks, but if a server rejects requests with blank Referer headers, it may reduce functionality for some users.

In practice, Referer headers may be removed by the browser, the operating system, or a network monitoring system for privacy issues. For example, if you click on a link to visit a website from a Google search, the website can know what Google search you made to visit its website from the Referer header. Some modern browsers also have options that let users disable sending the Referer header on all requests. Because not all requests are guaranteed to have a Referer header, it is usually only used as a defense-in-depth strategy in addition to CSRF tokens, instead of as the only defense against CSRF attacks.

Further reading: OWASP Cheat Sheet on CSRF

[1] Yes, the "Referer" field represents a roughly three decade old misspelling of referrer. This is a silly example of how "legacy", that is old design decisions, can impact things decades later because it can be very hard to change things