Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is **a web security vulnerability that allows an attacker to trick a user into performing unwanted actions on a trusted website where the user is authenticated**. CSRF exploits the trust that a website has in a user's browser, primarily **through the misuse of cookies for authentication**.

1. How CSRF Works

1. **Victim Authentication**

- **The victim logs into a trusted website** (e.g., example.com), which **sets an authentication cookie in their browser**.

2. **Attacker's Setup**

- **The attacker crafts a malicious link or form on their own website (or a compromised site) that sends a request to the trusted website**.

3. **Unintentional Action**

- The victim clicks the malicious link or visits the attacker's page, which **triggers a request to the trusted website**.
- Since the victim's browser automatically includes the authentication cookie with the request, **the trusted website processes it as if it came from the authenticated user**.

2. Why Cookies Are Involved

- Cookies are automatically sent with requests to the domain that set them, regardless of where the request originates.
- CSRF attacks exploit this behavior to send authenticated requests on behalf of the user without their knowledge.

3. Example of a CSRF Attack

a. Scenario: A Banking Website

- The victim is logged into bank.com and has an active session authenticated via cookies.

b. Attacker's Malicious Request

- The attacker hosts a page with the following code:

```
<form action="https://bank.com/transfer" method="POST">
  <input type="hidden" name="to" value="attacker_account">
  <input type="hidden" name="amount" value="1000">
  <input type="submit" value="Click me for a surprise!">
</form>
```

c. Victim Interaction

- The victim, while still logged into bank.com, visits the attacker's page and clicks the form submission button (or the form is submitted automatically via JavaScript).
- The request to https://bank.com/transfer includes the victim's valid cookies, and the server processes the transfer.

4. Mitigating CSRF

a. Use of CSRF Tokens

- **Include a unique CSRF token in each form or request that is validated by the server**.
- Example:
  - The server generates a token and embeds it in a form
  - The server validates the token when processing the form submission.

```
<input type="hidden" name="csrf_token" value="random_token123">
```

b. SameSite Cookies

- Use the **SameSite** attribute to restrict cookies from being sent with cross-site requests
  - Strict: Cookies are sent only for requests originating from the same site.
  - Lax: Cookies are sent for top-level navigation but not for background requests.
  - Example:

```
Set-Cookie: sessionId=abc123; SameSite=Strict
```

c. Require Authentication for Sensitive Actions

- Re-authenticate users or **require additional confirmation (e.g., OTP, password) for critical actions**.

d. CORS (Cross-Origin Resource Sharing)

- Configure servers to **only accept cross-origin requests from trusted origins**.

e. Verify HTTP Referer Header

- **Validate that the Referer header of incoming requests matches the trusted domain**.

5. Examples of Mitigation

a. Using CSRF Tokens in a Web App

- Backend (Python Flask Example):

```
from flask import Flask, request, render_template_string, session
import secrets

app = Flask(__name__)
app.secret_key = 'secret_key'

@app.route('/form', methods=['GET'])
def form():
    csrf_token = secrets.token_hex(16)
    session['csrf_token'] = csrf_token
    return render_template_string('''
    <form method="POST" action="/submit">
        <input type="hidden" name="csrf_token" value="{{ csrf_token }}">
        <input type="text" name="data">
        <input type="submit">
    </form>
    ''', csrf_token=csrf_token)

@app.route('/submit', methods=['POST'])
def submit():
    if request.form['csrf_token'] != session.get('csrf_token'):
        return "CSRF Attack Detected!", 403
    return "Form submitted successfully!"
```

b. Setting SameSite Cookies

- Example in Express.js (Node.js):

```
app.use(cookieSession({
  name: 'session',
  keys: ['key1', 'key2'],
  cookie: {
    secure: true,
    httpOnly: true,
    sameSite: 'Strict'
  }
}));
```

6. Why CSRF Is Dangerous

- **Exploitation of Trust**
  - The attack leverages the fact that servers inherently trust cookies sent by the browser.
- **Silent Execution**
  - The victim often has no idea their session has been hijacked or actions have been performed.
- **Broad Impact**
  - Any application that relies on cookies for authentication is susceptible if not properly protected.

7. Key Differences Between CSRF and XSS

Aspect	CSRF	XSS
Attack Vector	Exploits user trust in a website.	Exploits website trust in user input.
Exploited Mechanism	Relies on automatic cookie sending.	Injects and executes malicious scripts.
Primary Goal	Force unauthorized actions.	Steal data or execute malicious actions.
Mitigation	CSRF tokens, SameSite cookies.	Input validation, Content Security Policy (CSP).

8. Summary

Aspect	Details
What is CSRF?	A vulnerability where attackers trick users into performing unwanted actions on a trusted website.
How It Works	Exploits automatic inclusion of authentication cookies with cross-site requests.
Primary Defense	CSRF tokens, SameSite cookies, and referer validation.
Why It's Dangerous	Can lead to unauthorized actions, data theft, or account compromise.
Best Practices	Use CSRF tokens, restrict cookies with SameSite, and implement multi-factor authentication.

CSRF is a critical vulnerability that exploits the way cookies are automatically sent with requests. Implementing robust defenses, such as **CSRF tokens and SameSite cookies, is essential to secure web applications from this threat**. By understanding how CSRF works and its interaction with cookies, developers can build safer and more secure web applications.