# Lesson 6: Cross-Site Request Forgery (CSRF)

## Introduction

In previous lessons, we explored how browsers manage state across requests using cookies and how websites isolate themselves from each other using the Same-Origin Policy. Cross-Site Request Forgery (CSRF) is an attack that exploits how browsers handle cookies - specifically, that browsers automatically include cookies when making requests to a site, even when those requests are initiated by a different site.

## Understanding CSRF

CSRF is an attack that forces an end user to execute unwanted actions on a website where they are currently authenticated. The key insight is that when you are logged into a website, your browser will automatically include your authentication cookies with every request to that site, regardless of which site initiated the request.

For example, imagine you are logged into your bank at bank.com and then visit attacker.com. The attacker’s site could contain code that causes your browser to make a request to bank.com/transfer?to=attacker&amount=1000. Because you are logged in to bank.com, your browser will automatically include your authentication cookies with this request. The bank’s server sees a valid request with valid authentication cookies and has no reliable way to determine that you didn’t intend to make this request.

Common CSRF attacks include:

* Transferring funds from banking/financial sites
* Changing account email address or password
* Purchasing items from e-commerce sites
* Making posts or sending messages on social media
* For admin users: adding new admin accounts or executing privileged actions

## How CSRF Works

Let’s examine a concrete example. Consider this legitimate transfer form on bank.com:

<form method="POST" action="/transfer">  
 Send amount: <input name="amount" />  
 To user: <input name="to" />  
 <input type="submit" value="Send" />  
</form>

An attacker could create a malicious page on attacker.com containing:

<form method="POST" action="http://bank.com/transfer">  
 <input name="amount" value="1000" />  
 <input name="to" value="attacker" />  
</form>  
<script>  
 document.forms[0].submit(); // Auto-submit the form  
</script>

When a victim who is logged into bank.com visits the malicious page:

1. The attacker’s JavaScript automatically submits the form to bank.com
2. The browser includes the victim’s bank.com cookies with the request because cookies are automatically included in cross-origin requests
3. bank.com receives a valid request with valid authentication cookies
4. The transfer executes using the victim’s authenticated session

Importantly, this attack works *despite* the Same-Origin policy, because the attacker doesn’t need to be able to see the response from the forged request. They only need the side effects (like the transfer) to occur. The Same-Origin Policy prevents attacker.com from reading the response from bank.com, but it doesn’t prevent the request from being made in the first place.

## CSRF Mitigations

### The Referer Header

One approach is to check the HTTP Referer header, which indicates which URL initiated the request:

app.post('/transfer', (req, res) => {  
 const referer = req.headers.referer;  
 if (!referer || !referer.startsWith('https://bank.com')) {  
 res.status(403).send('Invalid referer');  
 return;  
 }  
 // Process transfer...  
});

However, this is not a reliable defense because:

1. The Referer header might be stripped by browsers or network proxies for privacy reasons
2. Users or browser extensions might disable sending the Referer header
3. Sites can opt out of sending the Referer header using the Referrer-Policy header
4. HTTP caches might serve cached responses without checking the Referer
5. The header can potentially be spoofed by attackers in some scenarios

### SameSite Cookies

A more robust solution is to use the SameSite cookie attribute, which tells browsers when they should include cookies in cross-site requests:

Set-Cookie: sessionId=abc123; SameSite=Lax

The SameSite attribute has three possible values: - None: Always send cookies (this was the default before 2020) - Lax: Send cookies for top-level navigations (like clicking links) but not for sub-resource requests (like images, frames, or form submissions) - Strict: Only send cookies if the request originates from the same site that set the cookie

For most applications, SameSite=Lax provides good security while maintaining functionality:

res.cookie('sessionId', sessionId, {  
 httpOnly: true, // Prevent JavaScript access  
 secure: true, // Require HTTPS  
 sameSite: 'lax', // CSRF protection  
 path: '/'  
});

With SameSite=Lax, the CSRF attack described earlier would fail because the browser would not include the victim’s bank.com cookies when attacker.com submits the form.

Some legitimate use cases might require SameSite=None (which must be paired with Secure):

* Social media embeds (like Facebook comments)
* Third-party payment processing widgets
* Single sign-on across different subdomains
* Third-party API integrations

If you need SameSite=None, you should implement additional protections like CSRF tokens.

### CSRF Tokens

A CSRF token is a random value that your server generates and includes in forms. When the form is submitted, the server verifies that the token in the request matches the one it generated. Since an attacker can’t predict the CSRF token, and can’t read it from the HTML on a page due to Same-Origin Policy, they will not be able to include the correct token in a forged form submission.

Here’s how to implement CSRF tokens:

// Generate token when rendering the form  
app.get('/transfer', (req, res) => {  
 const csrfToken = crypto.randomBytes(32).toString('hex');  
 // Store token in session  
 req.session.csrfToken = csrfToken;  
 res.send(`  
 <form method="POST" action="/transfer">  
 <input type="hidden" name="csrf" value="${csrfToken}" />  
 Amount: <input name="amount" />  
 To: <input name="to" />  
 <input type="submit" value="Send" />  
 </form>  
 `);  
});  
  
// Verify token when processing the request  
app.post('/transfer', (req, res) => {  
 if (req.body.csrf !== req.session.csrfToken) {  
 res.status(403).send('Invalid CSRF token');  
 return;  
 }  
 // Process transfer...  
});

### GET vs POST Requests

Using POST instead of GET for state-changing operations helps prevent CSRF in several ways:

1. Browsers only send GET requests when loading images or navigating to URLs. POST requests require either form submission or JavaScript’s fetch/XMLHttpRequest APIs. Therefore, requiring state-changing requests to be POST requests minimizes the ways in which an attacker can forge requests.
2. GET requests are easier to forge because they can be triggered by:

* <img src="http://bank.com/transfer?to=attacker&amount=1000">

1. GET requests might be logged by browser history, server logs, and proxy servers, potentially exposing sensitive data.
2. Some browsers and networks cache GET requests, which could lead to unintended repeated actions.

Therefore, while using POST doesn’t prevent CSRF on its own, it reduces the attack surface and follows proper HTTP semantics where GET requests should be “safe” and not cause state changes.

## Best Practices

1. Use SameSite=Lax cookies by default
2. Don’t rely solely on checking the Referer header
3. Use POST requests (not GET) for state-changing operations
4. Implement proper session management (as covered in Lesson 5)
5. Consider additional protections like CSRF tokens for critical operations
6. Always use HTTPS to prevent cookie theft through network attacks

Modern browsers default to SameSite=Lax, which helps prevent CSRF attacks. However, understanding CSRF remains important for:

* Supporting older browsers
* Dealing with legacy systems
* Implementing cross-origin features safely
* Understanding why certain security measures exist

In the next lesson, we’ll explore Cross-Site Scripting (XSS), another common web vulnerability that can bypass many CSRF protections.