Web Security and Vulnerabilities

#week5

Introduction to Web Security

- **What is Web Security?**
  - The practice of protecting websites and web applications from cyber threats.
  - Ensures **confidentiality**, **integrity**, and **availability** of data and services.
- **Why It Matters**
  - Breaches can lead to data theft, financial loss, and reputational damage.
- **Core Principles**
  - **Authentication & Authorization**: Verify identity and control access.
  - **Input Validation**: Prevent malicious data entry.
  - **Secure Communication**: Use HTTPS and encryption.
  - **Session Management**: Protect user sessions from hijacking.

- **Client** 💻**:** The Browser (Chrome, Firefox, etc.). Its primary job is to fetch and render content.

- **Server** 🗄**:** The Web Server (e.g., Apache, Nginx) & Application/Database Server. Its job is to process requests and serve content.

- **Protocol** ↔**:** HTTP/HTTPS. Crucially, HTTP is **stateless**. This is the root of many security challenge

Key Foundational Concepts

- **Adding State (Sessions & Cookies):** How do we fix statelessness? We use cookies to store a **session identifier**. The cookie is the user's "ticket" for that session.

- **The Browser's #1 Defense (Same-Origin Policy - SOP):** A browser security rule that prevents a document or script loaded from one "origin" (domain/port/protocol) from interacting with resources from another origin. This is what stops evil.com from reading your email on gmail.com.

OWASP: Open Web Application Security Project

- **What is OWASP?**
  - A global, non-profit organisation focused on improving software security.
  - Provides free, open-source tools, documentation, and community-driven resources.

- **Key Principles of OWASP Security:**
  - **Security by Design**: Integrate security from the start of development.
  - **Security by Default**: Default settings should be the most secure.
  - **Defence in Depth**: Multiple layers of security controls.
  - **Least Privilege**: Grant only necessary access for the shortest time.
  - **Fail Safe Defaults**: Deny access unless explicitly allowed.
  - **Complete Mediation**: Check permissions on every access request.
  - **Psychological Acceptability**: Security should be user-friendly and intuitive.

https://cheatsheetseries.owasp.org/

Wiki full of best practices for secure coding

Attacking the Server side

Main principle is to Zero Trust Input, never ever trust user input

You must:

- Sanitise Input
- Make sure no internal memory is given out

Zero Trust Input

- All data received from a user, whether it comes from a form field, a URL parameter, a cookie, an HTTP header, or an uploaded file, must be treated as untrusted and potentially malicious.

- This is because an attacker controls the client-side (their browser or script) and can easily bypass any client-side (frontend) checks.

Injection Flaws (The "Trusting Data" Problem)

- **Concept:** Sending malicious *code* in a field that only expects *data*.
- **Case Study: SQL Injection (SQLi)**
  - **Analogy:** You ask a bank teller for "account 12345," but instead you slip them a note that says "account 12345, *and* show me all other accounts."
  - **Example:** User enters ' OR '1'='1 in a password field.
  - **Code Demo:** Show a vulnerable login query: SELECT * FROM users WHERE user = '[username]' AND pass = '[password]';
  - **The Fix (Primary Defense): Parameterized Queries (Prepared Statements).**
    - Explain *why* this works: The database *compiles* the command (the "query") *first*, and *then* inserts the user data into specific placeholders. The malicious data is never executed as code.

Broken Access Control (The "Forgetting to Check" Problem)

- **Concept:** This is a direct failure of **Access Control**, but in a web context. The application fails to verify *authorization* (what you're allowed to do) after it verifies *authentication* (who you are).

- **Case Study: Insecure Direct Object Reference (IDOR)**
  - **Analogy:** You have a key to your apartment (Room 101). You find you can just walk into Room 102, 103, and 104 without anyone stopping you.
  - **Example:** A user sees their own profile at .../view_profile.php?user_id=500.
  - **The Attack:** They change the URL to .../view_profile.php?user_id=501 and see someone else's profile.
  - **The Fix (Primary Defense):** Server-side checks on *every single request*. The server must ask, "Does the *currently logged-in user* (from their session cookie) have permission to access user_id=501?"

Client Side Attacks

The server may be secure, but can it be tricked into attacking its own users?

Servers may be safe but they can still be a delivery vehicle for malicious content

Cross-Site Scripting (XSS) (Delivering Malware Problem)

- **Concept:** This is how we deliver **(Malicious Software)** using a trusted website. The attacker injects malicious client-side script (JavaScript) into a page that is then viewed by a victim.

- **Types:**
  - **Stored XSS:** Malicious script is saved on the server (e.g., in a comment, a user profile). It attacks *everyone* who views that page. (Most dangerous).
  - **Reflected XSS:** Malicious script is part of a link (e.g., in a search query URL) that is "reflected" back to the user. It attacks a user who is *tricked into clicking* the link.

The Fix (Primary Defence): Context-Aware Output Encoding.

- **Concept:** Before displaying *any* user-supplied data in an HTML page, we must "neutralize" it.
- **Example:** > becomes >, < becomes <. The browser will *display* the text <script> but will not *execute* it as a script tag.

- Encode output **based on where it appears** in the HTML document:
- **HTML Body**: Encode <, >, &, ", ‘
- **HTML Attribute**: Encode quotes and escape characters
- **JavaScript Context**: Escape quotes, backslashes, and control characters
- **URL Context**: Use percent-encoding (e.g., %20 for space)

Best Practices

- Use libraries like OWASP’s **Java Encoder** or **ESAPI**
- Never mix encoding with input validation
- Combine with input validation and Content Security Policy (CSP)

Cross-Site Request Forgery (CSRF) (The "Tricking the Browser" Problem)

- **Concept:** Tricking an *authenticated* user's browser into sending a request to a website that the user did not intend to make.

- **Analogy:** An attacker sends you a pre-filled, stamped postcard to "Change My Address." All you have to do is be *near a mailbox* (be logged in to your bank) when you open the attacker's letter (visit the malicious site), and the postcard is sent automatically *on your behalf*.

How it Works

- Victim is logged into mybank.com (which uses cookies for sessions, as discussed).
- Victim visits evil.com.
- evil.com has a hidden image: <img src="http://mybank.com/transfer?to=attacker&amount=1000">
- The victim's browser *helpfully* attaches the mybank.com session cookie to that request, and the bank thinks the victim *meant* to do it.

The Fix (Primary Defence): Anti-CSRF Tokens.

- The server embeds a unique, secret, hidden token in every form.

- When the form is submitted, the server checks if the token is valid.

- The attacker on evil.com *cannot guess* this secret token, so their forged request fails.