Web Security

Web security refers to the measures, practices, and technologies that are implemented to protect websites, web applications, and online services from various security threats and vulnerabilities. It ensures that the data, users, and servers involved in web-based interactions remain safe from malicious activities such as hacking, data breaches, malware attacks, and other cyber threats.

Types of Vulnerabilities

1. Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a security vulnerability that allows attackers to inject malicious scripts (usually JavaScript) into webpages viewed by other users. These malicious scripts can be used to steal sensitive data, impersonate users, or take control of the victim's interaction with the website.

XSS exploits vulnerabilities in a web application where it fails to properly filter or sanitize user input. When a victim visits a page with injected malicious code, the script runs in their browser, performing actions on behalf of the attacker, such as stealing cookies, sending data to an attacker-controlled server, or impersonating the user.

1.1 Reflected XSS:

This occurs when the malicious script is part of the user's HTTP request and is reflected back to them immediately in the server's response.

The attacker creates a URL with a malicious payload and convinces the victim to click on it. The website then reflects the script in the response without filtering it, causing the malicious code to execute in the victim's browser.

```
https://example.com/search?query=<img src="nonexistent.jpg"
onerror="alert('Hacked!');">
```

When the user clicks this URL, the onerror event triggers the alert() function, showing the "Hacked!" message.

1.2 Stored XSS

This happens when the malicious script is stored on the server (e.g., in a database) and then served to users who access the page later.

The attacker inputs the malicious script into a form or input field (like a comment box or a forum post), and the server stores this input without sanitizing it. Later, when another user views the page, the stored script is executed.

A user posts the following comment on a blog:

```
<img src="nonexistent.jpg" onerror="alert('Hacked!');">
```

When other users view this comment, the browser executes the onerror JavaScript.

How to prevent XSS attacks :

1. Filter and Validate Input

- Only allow expected data, like text or numbers. Block any harmful characters, such as <script>.
- For fields like comments or form inputs, clean or escape potentially dangerous HTML tags.

example :-

```
<p>Comment: &lt;script&gt;alert('XSS')&lt;/script&gt;</p>
```

2. Encode Output

- Always encode special characters (<, >, &) to prevent them from being executed.
- Ensure user data inside JavaScript is encoded to avoid execution.

example :-

```
echo htmlspecialchars($user_input, ENT_QUOTES, 'UTF-8');
```

3. Avoid Using innerHTML

- innerHTML can execute embedded JavaScript.
- Use textContent or innerText to insert plain text instead of HTML..

example :-

```
document.getElementById('output').textContent = user_input;
```

4. Use Content Security Policy (CSP)

- Set CSP to only allow scripts from trusted sources.
- Block inline JavaScript by using 'self' in the CSP header.

example :-

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

5. Sanitize HTML Input

- Use libraries like DOMPurify to safely sanitize user input before rendering it on the page.

6 . HTTP only Cookies

- HTTP Only cookies can't be accessed by client side JavaScript. So session id or tokens can't be stolen in a XSS attack

```
Set-Cookie: sessionid=abc123; HttpOnly; Secure; SameSite=Strict
```

2. CSRF Attacks

Cross-Site Request Forgery (CSRF) is a type of security attack where a malicious actor tricks a user into performing actions on a website without their consent. The attacker makes unauthorized requests on behalf of the victim, leveraging their authenticated session to carry out harmful actions such as changing passwords or transferring money to an account etc.

Example of a CSRF Attack:

Imagine you are logged into your bank account and have an active session. At the same time, you visit a malicious website that has embedded a request to transfer money from your account to the attacker's account.

1. You visit the malicious website.
2. The website silently sends a request to the bank's server, for example:
POST /transferFunds?to=attackerAccount&amount=1000
3. Because you are still logged into the bank, your browser automatically sends the request using your credentials (cookies, session tokens).
4. The bank processes the transfer as if it was you who initiated the request.

Even though you didn't intend to make that transfer, your browser sent the request because it trusted the active session with the bank.

How to Prevent CSRF Attacks?

1. Use Anti-CSRF Tokens:

To prevent CSRF attacks, generate a unique token for each session and include it in forms and requests. When a request is made, the server checks if the token in the request matches the one it originally generated. If the tokens don't match, the request is rejected. This method ensures that the request is coming from the user, thus preventing unauthorized requests from other sources.

2. SameSite Cookie Attribute:

The SameSite cookie attribute can be used to prevent browsers from sending cookies with cross-origin requests (requests coming from a different site). By setting the SameSite=Lax or SameSite=Strict attribute on cookies, you can ensure that cookies are not sent along with malicious requests from other websites. For example:

```
Set-Cookie: sessionid=your_session_id; SameSite=Strict
```

3 .Check Referrer or Origin Headers:

Another method of protection involves validating the Referrer or Origin headers in requests. These headers indicate the origin of the request, and by checking that they match your website's domain, you can ensure that the request is coming from a legitimate source. If these headers are missing or don't match the expected domain, you can block the request to prevent CSRF

3 .SQL Injection

SQL Injection (SQLi) is a type of attack where an attacker manipulates a web application's database query by inserting malicious SQL code into an input field (e.g., a form or URL parameter). The goal of SQL injection is to interfere with the database's operations, potentially allowing attackers to view, modify, or delete data, bypass authentication, or execute administrative operations on the database.

Example of an SQL Injection Attack:

Imagine a website has a login form where users enter their username and password. The backend might construct a query like this:

```
SELECT * FROM users WHERE username = 'USER_INPUT' AND password = 'USER_INPUT';
```

If an attacker enters the following into the username field:

```
' OR '1' = '1
```

The query becomes:

```
SELECT * FROM users WHERE username = '' OR '1' = '1' AND password = 'USER_INPUT';
```

The condition '1' = '1' is always true, so the query will return a valid result, bypassing authentication and allowing the attacker to log in without a valid username or password.

How to Prevent SQL Injection Attacks:

Use Prepared Statements (Parameterized Queries):

Prepared statements ensure that user input is treated as data, not executable SQL code. With this approach, user inputs are bound to query parameters rather than inserted directly into the SQL query. This prevents attackers from altering the query's structure. Most modern programming languages and databases (e.g., PHP, Python, Java, MySQL, PostgreSQL) support prepared statements, making this a best practice

```
connection.execute('SELECT * FROM users WHERE username = ? AND password = ?', [username, password], (err, results) => { if (results.length > 0) console.log('Authenticated'); else console.log('Invalid credentials'); });
```

4. Weak Password Hashing

Password Hashing and Salting are techniques used to securely store user passwords. Hashing transforms the password into a fixed-length string (using algorithms like SHA-256 or bcrypt) that can't be easily converted back to the original password. Salting adds a random string of characters (a "salt") to the password before hashing to ensure that identical passwords have different hash values.

When these methods are not used, it creates a vulnerability that attackers can exploit, such as rainbow table attacks or brute force attacks, where attackers can easily guess or pre-compute hashed values for common passwords.

Example of the LinkedIn Data Breach:

In 2012, LinkedIn suffered a data breach that exposed 6.5 million hashed passwords. Although the passwords were hashed, they were hashed using SHA-1 without salting. This made it easier for attackers to perform rainbow table attacks, where precomputed tables of hash values for common passwords are used to reverse the hashes back into the original passwords.

In 2016, LinkedIn's data breach was re-exposed, as the attackers released more than 100 million stolen credentials (including usernames and hashed passwords). Because the hashes were weak and unsalted, attackers could easily crack a significant number of passwords. The breach was a reminder of the importance of salted hashes and strong hashing algorithms like bcrypt

How to Prevent Weak Password Hashing:

1. Use a Secure Hashing Algorithm (e.g., bcrypt, PBKDF2, Argon2):

A good hashing algorithm ensures that even if attackers know the hashing method, they cannot easily reverse the password. These algorithms are slow, making brute-force attacks more difficult.

2. Salt the Passwords:

Salt is a random value added to the password before hashing, ensuring that even if two users have the same password, their hashed passwords are different.
example:

```
const bcrypt = require('bcrypt');
bcrypt.hash('password123', 10, (err, hashedPassword) => { if (!err)
console.log('Hashed Password:', hashedPassword); });
```