Reconnaissance (Information Gathering)

1. Passive Reconnaissance (No direct interaction with the target):

Techniques and Tools:

- **Google Dorking**: Leverage advanced search operators to find hidden information on the web.
    - Example:
        - `site:example.com intitle:"index of"`: Search for index pages within the target domain.
        - `filetype:pdf site:example.com confidential`: Search for PDF documents labeled as confidential.
- **WHOIS Lookups**: Use the `whois` command to find details about domain registration, ownership, and IP addresses.
    - Example: `whois example.com`
- **DNS Enumeration**: Tools like `dig` or `nslookup` help gather DNS records.
    - Example: `dig example.com any` for detailed DNS records.
- **Subdomain Enumeration**: Tools like **Sublist3r** or **Amass** help in finding subdomains.
    - Example: `sublist3r -d example.com`
- **TheHarvester**: Useful for collecting emails, subdomains, and more from various data sources like search engines, social media, and PGP servers.
    - Example: `theharvester -d example.com -b google`
- **Shodan**: Use Shodan to find devices, services, and potential vulnerabilities exposed by the target.
    - Example: `shodan host <target-ip>`
- **Netcraft**: Can be used to gather information about hosting infrastructure, server technologies, and past domain history.

2. Active Reconnaissance (Interacts with the target):

This step involves interacting directly with the target system to probe its defenses. These activities, while more detectable, provide deeper insights into the target's infrastructure.

Techniques and Tools:

- **Nmap**: Essential for port scanning, service enumeration, and fingerprinting open services.
    - Example: `nmap -sS -sV -p 1-65535 example.com` to perform a full TCP scan and service detection.
- **Web Application Scanners**: Tools like **Nikto** or **Gobuster** help to find directories, web vulnerabilities, and files.
    - Example: `nikto -h http://example.com` to scan for web server vulnerabilities.

    - Example: `gobuster dir -u http://example.com -w /path/to/wordlist.txt` for directory brute-forcing.
- **Banner Grabbing**: Extract information about services running on open ports.
    - Example: Use `telnet example.com 80` or `curl -I http://example.com` to manually grab HTTP headers.
- **DNS Zone Transfer**: Check for misconfigured DNS that might allow zone transfer.
    - Example: `dig axfr @ns1.example.com example.com`

Identify Security Misconfigurations (P1-P5):

1. Initial Information Gathering:

- **Tools**: Nmap, Nikto, or OWASP ZAP.
- **Command (Nmap)**: `nmap -sV -T4 -A target.com`
    - This command helps map open ports and services, identifying potential misconfigurations in running services and unnecessary open ports.

2. Scan for Default Credentials and Exposed Services:

- **Tools**: Burp Suite, Nessus.
- **Technique**: Use automated scanners like Nessus to identify default credentials and check for improperly configured services such as open databases, FTP servers, or admin panels.

3. Security Header Configuration:

- **Tools**: SecurityHeaders.io or Burp Suite.
- **Command**: Using Burp Suite, intercept requests and inspect response headers to check if critical security headers (Content-Security-Policy, HSTS, X-Frame-Options, etc.) are present.
- **Technique**: Analyze the headers to ensure protections like HTTP Strict Transport Security (HSTS) are correctly applied.

4. Check for Outdated Software and Libraries:

- **Tools**: OWASP Dependency-Check, Acunetix.
- **Technique**: Use tools to scan for outdated libraries or software that may contain known vulnerabilities.
- **Command (OWASP Dependency-Check)**: `dependency-check --project WebApp --scan /path/to/application`

        - This tool scans the application dependencies for known vulnerabilities, helping identify security risks stemming from outdated libraries.

5. Cloud Storage and API Misconfigurations:

- **Tools**: AWS CLI, ScoutSuite.
- **Technique**: For cloud environments, use tools like ScoutSuite to audit AWS, GCP, or Azure configurations. Look for misconfigured S3 buckets or overly permissive API endpoints.
- **Command (AWS CLI)**: `aws s3api get-bucket-acl --bucket bucket-name`
    - This command helps you verify the access control list (ACL) of S3 buckets, ensuring they are not publicly accessible unless intended.

6. File Permissions and Directory Listing:

- **Tools**: Nikto, Dirb.
- **Command (Nikto)**: `nikto -h` target.com
    - This will scan the web server for vulnerabilities, including directory listing enabled or improper file permissions that could expose sensitive files.

7. Testing for Unpatched Software and Misconfigured Firewalls:

- **Tools**: OpenVAS, Nessus.
- **Command (OpenVAS)**: `openvas-start`
    - Once the scan is complete, look at reports for systems running outdated software or improper firewall configurations that leave unnecessary ports exposed.

8. Configuration Management:

- **Tools**: Ansible, Puppet.
- **Technique**: These tools help ensure consistent configurations across systems, reducing the chance of human error or misconfigurations. You can write playbooks to enforce secure configurations.
- **Command (Ansible)**: `ansible-playbook secure_config.yaml`
    - A YAML playbook can automate the application of security settings across multiple systems.

Testing for Injection Attacks (P1-P5):

1. Preparation:

- **Understand the Target:** First, identify areas of the application that involve interaction with databases, file systems, or external services where injection could occur. These typically include login forms, search fields, file uploads, and API calls.
- **Tools Setup:** Set up tools such as **Burp Suite**, **OWASP ZAP**, and **SQLMap** to automate parts of the process. For command injection, tools like **Commix** are highly useful.

2. Input Validation Testing:

- **Manual Exploration:** Manually identify fields accepting user input (e.g., forms, URL parameters, cookies). Use a web proxy like Burp Suite to intercept requests and manipulate them.
- **Basic Fuzzing:** Start with simple fuzzing techniques, injecting common payloads (e.g., `' OR 1=1 --` for SQL injection) to see how the application responds. Use tools like **Wfuzz** to automate this process and test multiple input vectors.

3. SQL Injection Testing:

- **Automated SQLi Testing:** Use **SQLMap** to automate SQL injection detection. Start by passing vulnerable parameters (intercepted via Burp) to SQLMap for injection testing.
- **Manual SQLi Testing:** Inject basic SQL payloads into vulnerable input fields, like `' OR '1'='1`. Watch for database errors or unauthorized access, indicating a vulnerability.
- **Advanced SQLi:** Test for blind SQL injection using time-based or boolean techniques by injecting payloads such as `AND SLEEP(5)` and checking for time delays.

4. Command Injection Testing:

- **Manual OS Command Injection:** Identify input fields that might interact with the operating system (e.g., file upload names, shell commands). Inject payloads like `; ls -la` or `&& whoami` to see if system-level commands are executed.
- **Automated Testing:** Use **Commix** to automate command injection testing by passing vulnerable URLs or forms and allowing the tool to inject common command payloads.

5. Testing for Cross-Site Scripting (XSS) Injection:

- **Manual XSS Testing:** Insert script payloads into input fields to check if user-supplied data is rendered without proper encoding. For example, inject `"<script>alert('XSS')</script>"` into forms or URL parameters.
- **Automated XSS Testing:** Tools like **OWASP ZAP** or **XSStrike** can be used to scan for reflected or stored XSS automatically across various endpoints.

6. Server-Side Template Injection (SSTI):

- **Payload Testing:** If the application uses templating engines, inject payloads like `{{7*7}}` to see if the input is executed. If the result shows `49`, it indicates a vulnerability.
- **Advanced SSTI Scanning:** Use tools like **TplMap** to detect and exploit SSTI vulnerabilities automatically.

7. LDAP/XXE Injection Testing:

- **LDAP Injection:** Test input fields that interact with directory services by injecting LDAP payloads, such as `*)(uid=*))(|(uid=*`.
- **XML External Entity (XXE) Testing:** Inject malicious XML data like `<!ENTITY xxe SYSTEM "file:///etc/passwd">` into XML payloads sent to the server.

8. Utilize Logging and Error Responses:

- Monitor server responses and logs for errors or anomalies that reveal potential vulnerabilities during your tests. Injection flaws often cause exceptions or detailed error messages.

Authentication and Session Management Testing (P1-P5):

1. Test for Weak or Predictable Credentials (P1-P5)

- **Objective**: Verify that users cannot authenticate with weak or easily guessable credentials.
- **Technique**: Attempt brute force or dictionary attacks using common password lists (e.g., `rockyou.txt`).
- **Tools**:
    - **Hydra**: A tool for brute force attacks.
        - hydra -l admin -P rockyou.txt <IP> http-form-post "/login.php:username=^USER^&password=^PASS^:F=Invalid username or password"
    - **Burp Suite Intruder**: Use Burp Suite's Intruder to test credential stuffing and weak passwords automatically.

2. Test for Session Management Vulnerabilities (P1-P5)

- **Objective**: Ensure that session tokens (IDs) are securely generated, managed, and invalidated properly.
- **Tests**:
  - **Session ID Strength**: Verify entropy of session IDs by analyzing token randomness with Burp Suite's **Sequencer** tool.
  - **Session Fixation**: Ensure the session ID is regenerated upon login.
  - **Session Hijacking**: Check if tokens are sent over insecure channels (HTTP instead of HTTPS).
  - **Logout Functionality**: Verify that sessions are invalidated correctly on logout.
    - curl -I http://example.com
      Look for **Set-Cookie** headers to examine session token attributes (e.g., Secure, HttpOnly flags).
- **Tools**:
  - **Burp Suite**: Use the **Session Management** section to manually or automatically analyze session cookies.
  - **OWASP ZAP**: Provides similar capabilities to Burp Suite for testing session handling and token security.

3. Test for Insecure Authentication Mechanisms (P1-P5)

- **Objective**: Ensure proper implementation of authentication methods like multi-factor authentication (MFA) and OTP.
- **Tests**:
  - **2FA/MFA Bypass**: Test if you can bypass 2FA by analyzing the application's response to invalid codes or reusing valid session tokens.
  - **Automated Login Testing**: Using tools like **Postman**, create a request to simulate login attempts:
    - POST /api/login
      ```
      {
        "username": "testuser",
        "password": "password123"
      }
      ```
  - **Password Recovery**: Test for improper handling of recovery tokens.
- **Tools**:
  - **Postman** for API-based testing.
  - **Selenium** for automating UI tests (e.g., simulating login forms).

4. Test for Session Timeout and Logout Issues (P2-P5)

- **Objective**: Ensure sessions expire after inactivity and cannot be reused post-logout.
- **Tests**:

    - Set up an automated test using **Selenium** to check if session timeout works properly after the configured inactivity period.
    - Manually test by logging out and trying to reuse the old session token.
    - Confirm if sessions are invalidated properly across multiple devices and browsers.
- **Commands**:
    - curl --cookie "session_id=xyz" http://example.com/protected_page
      This helps you check if the old session token still provides access.
- **Tools**:
    - **Burp Suite**'s **Repeater** can be used to replay expired tokens.

5. Cross-Site Request Forgery (CSRF) Testing (P2-P4)

- **Objective**: Ensure that requests are properly validated to prevent CSRF attacks.
- **Tests**:
    - Verify that CSRF tokens are unique and included in every POST/PUT request.
    - Create a CSRF exploit proof of concept using Burp Suite's **CSRF PoC generator**.
    - Test for token reuse issues or lack of CSRF protection in sensitive functions like password changes.
- **Tools**:
    - **OWASP ZAP** for CSRF vulnerability scanning.
    - **Burp Suite Pro** for generating CSRF payloads.

6. Automated Testing (P1-P5)

- **Tools**:
    - **OWASP ZAP** and **Burp Suite** can automate session and authentication testing across multiple scenarios.
    - **Nessus** and **Qualys**: Use these scanners to identify common misconfigurations in authentication and session handling.

Testing for Cross-Site Scripting (XSS) (P1-P5):

1. Identify Input Points:

- Begin by identifying all user input points, such as search fields, URL parameters, comment boxes, forms, and even file upload fields, where data is submitted to the application and reflected back. These are common areas for XSS vulnerabilities to be introduced.
- Use web crawlers, like **Burp Suite's Spider** or **OWASP ZAP**, to automatically detect all input points that the application accepts.

2. Crafting XSS Payloads:

- Start with basic payloads to test if inputs are sanitized or not:
  - <script>alert('XSS');</script>
  - <img src="x" onerror="alert(1)">
- If the script is stripped or filtered, modify the payload to bypass basic filters:
  - <svg onload="alert(1)">
  - <iframe src="javascript:alert(1)">
- For **DOM-based XSS**:
  - <img src=x onerror=alert(123)>
- For reflected XSS, try injecting payloads into URL parameters.

3. Manual and Automated Testing:

- **Manual Testing**:
  - Insert XSS payloads into identified input fields and check if they are executed by the browser.
  - Inspect page source and browser console for any reflected or executed script. Use developer tools like **Chrome DevTools** to monitor requests and responses.
  - Example tools: **Burp Suite**, **Postman**, or even **curl** for sending HTTP requests:
    - curl "http://example.com/search?query=<script>alert('XSS')</script>"
- **Automated Testing**:
  - Tools like **OWASP ZAP**, **XSStrike**, or **Acunetix** can automatically scan and inject payloads across various input fields.
  - Configure automated tools to generate reports with details about XSS vulnerabilities and proof of concept (PoC) payloads.

4. Bypassing Filters:

- If initial payloads are blocked, attempt evasion techniques like encoding or using alternative tags:
  - Hover over me
  - <img src=x onerror=alert(String.fromCharCode(88,83,83))>

- Use **URL encoding** or **hex encoding** to bypass input validation:
    - %3Cscript%3Ealert(1)%3C%2Fscript%3E
- Specially crafted payloads, like base64-encoded scripts, can also be used:
    - <meta http-equiv="refresh" content="0;url=data:text/html;base64,...">

5. Tools for XSS Testing:

- **Burp Suite**: Use the **Intruder** tool to automate payload insertion across multiple fields and test for XSS.
- **OWASP ZAP**: This open-source tool allows for both manual and automated XSS testing, crawling the site and injecting malicious payloads.
- **XSStrike**: A tool specifically designed for fuzzing XSS payloads, attempting to bypass filters and generating various payloads.
- **XSSer**: An automatic tool designed to detect, exploit, and report XSS vulnerabilities in web applications.

6. Advanced Techniques:

- Look for **attribute-based XSS** vulnerabilities, where attackers use attributes like `onload`, `onerror`, or `onclick` to inject malicious scripts:
    - <img src="nonexistent.jpg" onerror="alert(document.cookie)">
- Explore **filter bypasses** using mixed case (e.g., `JaVaScRiPt`), or injecting payloads into unexpected locations (e.g., headers, user-agent strings).

Broken Access Control (BAC) (P1-P5):

1. Preparation & Reconnaissance

- **Identify the Scope**: Determine which functionalities or resources require access control (e.g., admin panels, user-specific data).
- **OSINT & Mapping**: Use tools like Burp Suite, ZAP, or manual exploration to map all accessible URLs, endpoints, and roles. Gather potential user roles and functionalities tied to access control.

---

2. Insecure Direct Object Reference (IDOR) Testing

- **IDOR Overview**: IDOR vulnerabilities occur when user-provided identifiers (e.g., user IDs, file IDs) are not properly validated against access control rules.
- **Manual Testing**: Using Burp Suite or ZAP, intercept requests that involve sensitive resource identifiers (e.g., `GET /profile?id=123`). Manually alter the `id` parameter

and observe if unauthorized access is granted (e.g., switching to `id=124` to access another user's data).
- **Automation**: Use Burp Suite's intruder to automate IDOR detection by brute-forcing various ID values in your intercepted request.

---

3. Forced Browsing & URL Manipulation

- **Test URL Restrictions**: Forced browsing occurs when an attacker bypasses access control checks by directly accessing URLs intended for privileged users (e.g., https://example.com/admin).
- **Tools**: Use Burp Suite or OWASP ZAP to crawl the site structure and test access to URLs that should be restricted (such as admin pages). Check for the absence of redirects or authentication challenges.
- **Path Traversal**: Test for file access vulnerabilities using relative paths (`../../etc/passwd`) to access protected directories or files.

---

4. Role-Based Access Control (RBAC) Validation

- **Role Manipulation**: Log in as users with different roles (e.g., regular user vs. admin) and capture requests. Modify the roles in cookies, headers, or parameters to verify if role enforcement is in place.
- **Session Token Manipulation**: Try swapping session tokens between users to check if session-based access controls are functioning properly.
- **Tools**: Use Burp Suite's Repeater feature to modify session tokens and role-related parameters in requests.

---

5. Access Control Matrix Testing

- **Define an Access Control Matrix**: Create a matrix listing each role (e.g., admin, user, guest) and corresponding actions they are permitted to perform. Verify if each action is properly restricted.
- **Check for Role Privilege Escalation**: Test if lower-privileged users (e.g., guests) can perform actions meant for higher-privileged users (e.g., admins).

---

6. Vulnerability Scanning

- **Automated Tools**: Use scanners like Burp Suite Pro, OWASP ZAP, and Nessus to identify common misconfigurations and BAC-related issues (such as missing authentication checks, improper session handling, or file permission issues).
- **Cross-Site Request Forgery (CSRF)**: Ensure that protected functions (like password changes) are not vulnerable to CSRF by verifying if CSRF tokens are present and validated.

---

7. Exploiting BAC Vulnerabilities

- **Privilege Escalation**: Attempt to elevate privileges by accessing admin functionality while logged in as a regular user.
- **Vertical Access Control**: Use manual testing or Burp Suite's automated tools to check if lower-privileged users can perform administrative tasks.
- **Horizontal Access Control**: Test whether users can access other users' data by altering request parameters (e.g., changing user IDs in the URL).

---

8. Post-Exploitation

- **Session Persistence**: Test if sensitive sessions (e.g., admin sessions) persist after logout, or if they are invalidated properly.
- **Data Leakage**: After gaining unauthorized access, verify the extent of sensitive data exposed (e.g., by viewing other users' personal information or modifying important data).

Sensitive Data Exposure (P1-P5):

1. Setup Phase:

- **Prepare the Environment**: Ensure you have permissions to test the target application. Update and configure your tools to ensure they're ready for vulnerability scanning.
- **Tools**: Tools like **Burp Suite**, **OWASP ZAP**, **Nessus**, and **Wireshark** are essential for capturing data and inspecting requests to identify sensitive data exposure.

2. Initial Information Gathering:

- **Analyze Data Flows**: Understand how data is transmitted between clients and servers. Look for areas where sensitive information (e.g., PII, session tokens) is sent over unprotected channels (HTTP instead of HTTPS).
- **Command**: Use `nmap` to identify open ports and services that may be handling sensitive data insecurely:
    - nmap -sV -p- target-ip
- **Capture Traffic**: Use **Wireshark** to monitor network traffic and detect if sensitive data (like session cookies or authentication tokens) is being transmitted in plain text.
    - tshark -i eth0 -Y "http.request"

3. Scanning for Vulnerabilities:

- **Automated Vulnerability Scanning**:
    - Use **Nessus** or **Burp Suite** to scan for common sensitive data exposure vulnerabilities (e.g., lack of encryption, unmasked credit card numbers). Set scans to detect insecure storage in local storage/session storage.
    - Example command in **Nessus**:
        - nessus -T xml -q -x target-ip
- **Test Token and Key Exposure**: Check if sensitive tokens (such as API keys, OAuth tokens) are exposed in URLs or referer headers:
    - Burp Suite > Proxy > HTTP History > Look for sensitive tokens in URL or referer.

4. Manual Testing for Sensitive Data Exposure:

- **Check for Unencrypted Data**: Manually check if sensitive data (like credit card numbers, social security numbers) is being stored in plaintext or is exposed in URLs.
    - **Example**: Inspect URLs for any tokens or sensitive information:
        - curl -I http://example.com/userdata?token=123456789
- **Cookie and Session Management**:
    - Test if sensitive cookies have proper flags set (`Secure`, `HttpOnly`, `SameSite`). Use **Burp Suite** to inspect cookie attributes.
    - Example using **cURL** to check response headers:

        - curl -I https://example.com/login

5. Test for Data Exposure in Source Code:

- **Analyze JavaScript and HTML**:
    - Inspect if sensitive data is hardcoded in JavaScript files, particularly in global variables or functions.
    - **Command**:
        - wget --mirror --convert-links --adjust-extension --page-requisites --no-parent http://example.com
        - grep -r "API_KEY" example.com/

6. Analyze Logs and Error Pages:

- Check for detailed server error messages or debug pages that might leak sensitive information (e.g., stack traces with database credentials).
    - Use **Burp Suite** or **OWASP ZAP** to capture error responses.

7. SSL/TLS Misconfigurations:

- Test for weak SSL/TLS configurations that could lead to exposure of sensitive data during transmission.
- Use tools like **SSL Labs** or **OpenSSL** to check the strength of the encryption.
    - openssl s_client -connect example.com:443

8. API Testing:

- **Test API Endpoints**: Look for exposed sensitive data in API responses. This includes PII or tokens being returned in API calls.
- Use **Postman** or **Burp Suite** to inspect API responses for sensitive data leakage.

9. Sensitive Data Storage:

- Check if sensitive data is being stored unencrypted in local or session storage on the client side. This can be checked using the browser's Developer Tools:
    - Chrome > Developer Tools > Application > Local Storage/Session Storage.

Denial-of-Service (DoS) vulnerabilities (P2-P5)

1. Identify the Attack Surface:

- Start by analyzing which services or components could be potential targets for DoS attacks, including the network (e.g., bandwidth consumption), application (e.g., resource exhaustion), and protocol layers (e.g., SYN flooding or UDP floods).
- For example, on a web application, check endpoints that might be vulnerable to high request volumes, or services like DNS or FTP that could be targets for protocol-specific attacks.

2. Simulate Network Layer Attacks:

- **SYN Flood Attack**: This is one of the most common DoS methods where the attacker sends many SYN requests but doesn't complete the handshake, overwhelming the target with half-open connections.
- **Command (using `hping3`):**
    - sudo hping3 --flood -S target.com -p 80 --rand-source
      This command will send SYN packets to port 80 at a very high rate, randomly varying the source IP to simulate a distributed attack

3. Simulate Application Layer Attacks (Layer 7):

- **Slowloris Attack**: A "low and slow" method that keeps multiple connections open by sending HTTP requests very slowly, gradually exhausting the target's resources.
- **Command (using slowloris.py)**:
    - python slowloris.py target.com
      This will send partial HTTP requests to the target, causing the server to hold the connection open and eventually become unresponsive(

- **HTTP Flood Attack**: This attack involves sending a large number of HTTP GET/POST requests to overwhelm the server's ability to process legitimate traffic.
- **Command (using `hping3`)**:
    - sudo hping3 -S --flood -p 80 target.com -i u1000
      This command sends a flood of HTTP requests, attempting to exhaust the server's resources

4. Simulate Protocol Layer Attacks:

- **UDP Flood Attack**: Sends massive volumes of UDP packets to a target, overwhelming the network or the service (e.g., DNS).
- **Command (using `hping3`)**:
    - sudo hping3 --flood --rand-source -p 53 -d 1024 target.com -u
      This sends a flood of UDP packets to port 53 (commonly used by DNS), with random source IPs

5. Test Rate Limiting Protections:

- After testing different attack types, check if the server or application has rate-limiting mechanisms in place. This can be verified by adjusting the request rate using tools like `nping`:
    - nping --tcp-connect -rate=10000 -c 500000 target.com

This sends TCP connection requests at a defined rate to test how the server handles multiple connections

Cryptographic Weaknesses Testing (P2-P5):

1. Check for Weak SSL/TLS Configurations

- **Objective**: Identify weak SSL/TLS ciphers, protocols, or certificate issues that can compromise data transmission.
- **Tools**:
    - **SSL Labs**: Use SSL Labs to perform an SSL/TLS configuration check.
    - **sslyze**: This is a Python tool for fast and comprehensive TLS configuration testing. You can install it with:
        - pip install sslyze
        - sslyze --regular example.com
    - **testssl.sh**: A bash script that tests SSL/TLS configurations on any given domain. You can run:
        - ./testssl.sh https://example.com
    - **openssl**: Manually check SSL configuration by initiating a connection:
        - openssl s_client -connect example.com:443
- **What to Look For**:
    - Weak ciphers (e.g., RC4, MD5-based encryption) and outdated protocols (e.g., SSLv2, SSLv3).
    - Use of weak certificates (less than 2048-bit keys, expired certificates)

2. Examine Cryptographic Algorithm Strength

- **Objective**: Ensure only secure, industry-standard algorithms (AES, RSA, etc.) are used for encryption.
- **Tools**:
    - **OWASP Wycheproof**: This tool provides test vectors to check the security of cryptographic algorithms in your code. You can download and run it to test implementations of RSA, ECC, AES, etc.
        - More details and examples: Wycheproof.
    - **CodeQL**: Perform static analysis on your codebase to detect cryptographic flaws. Install and run it with:
        - codeql database create codeql-db --language=java --source-root .

        - codeql query run --database=codeql-db query-file.ql
- **What to Look For**:
    - Use of outdated or vulnerable algorithms (e.g., DES, 3DES, Blowfish).
    - Weak or predictable random number generation (e.g., using `rand()` instead of cryptographically secure functions)

3. Check for Improper Key Management

- **Objective**: Ensure cryptographic keys are securely generated, stored, and managed.
- **Tools**:
    - **gpg**: Generate strong cryptographic keys using GPG:
        - gpg --gen-key
    - **OpenSSL**: Check key generation entropy and strength:
        - openssl rand -base64 32
- **What to Look For**:
    - Weak key lengths (less than 2048 bits for RSA).
    - Reused nonces, predictable PRNG seeds.
    - Keys stored in plaintext or hardcoded in the application

4. Validate Hashing and Salting Implementations

- **Objective**: Ensure proper salting and hashing techniques are applied to sensitive data like passwords.
- **Tools**:
    - **bcrypt**: A tool for secure password hashing. You can hash passwords like this:
    - **hashcat**: A password recovery tool that helps test the strength of your hashed passwords by cracking them.
- **What to Look For**:
    - Ensure passwords are hashed with a salt and a modern hashing algorithm (e.g., bcrypt, scrypt, PBKDF2) rather than MD5 or SHA1

5. Test for Cleartext Transmission of Sensitive Information

- **Objective**: Ensure all sensitive information is encrypted during transmission.
- **Tools**:
    - **Wireshark**: Monitor network traffic for cleartext transmission of sensitive data:
        - sudo wireshark
    - **curl**: Manually test API calls or data transmissions for secure protocols:
        - curl -kis http://example.com/api/login
- **What to Look For**:
    - Credentials or sensitive data sent via HTTP instead of HTTPS

6. Test Pseudo-Random Number Generator (PRNG)

- **Objective**: Ensure the random number generators used are cryptographically secure.
- **Tools**:
    - **dieharder**: A random number generator tester.
        - sudo apt-get install dieharder
        - dieharder -a -g 202
- **What to Look For**:
    - Predictable seed values in PRNG or reuse of the same seed, leading to weak cryptographic operations

Cross-Site Request Forgery (CSRF) (P2-P5):

1. Understand the Attack

- CSRF attacks force users to execute unintended actions on a web application where they are authenticated. These are state-changing requests, such as transferring funds or updating profile data, but the attacker doesn't see the response.
- The vulnerability occurs because browsers automatically include session cookies with every request, regardless of origin. This makes it hard for a web application to differentiate between a legitimate and forged request.

2. Test for CSRF Vulnerabilities Manually

Identify Target Forms and URLs:

- Look for any state-changing actions (e.g., password change, fund transfers) in the application. For instance, a URL might look like:
    - GET https://bank.com/transfer?amount=100&accountNumber=123456
- Build an exploit URL that modifies this request, tricking the victim into executing a malicious action:
    - GET https://bank.com/transfer?amount=10000&accountNumber=456789

Forge Requests in HTML Forms:

- You can embed a malicious request in an HTML element such as an image or form:
    - <img src="https://bank.com/transfer?amount=10000&accountNumber=456789" width="0" height="0">
- Send the link to the victim or embed it in a website they frequently visit to trigger the action automatically.

Validate Tokens (if present):

- If you notice CSRF protection (anti-CSRF tokens) in forms, try to extract and reuse the token.

    - var csrfToken = document.getElementsByName('csrf_token')[0].value;
- You can automate token extraction and submission using browser developer tools or a custom script.

3. Bypass CSRF Protections

- If an application uses weak tokens (e.g., predictable or static tokens), attempt brute-forcing token values using Burp Suite's **Intruder** or **Repeater**.
- If there is an XSS vulnerability on the same page, you can use it to steal and reuse the CSRF token, effectively bypassing protection measures.

4. Automated CSRF Testing Tools

- **OWASP ZAP**: It includes a built-in CSRF scanner that detects missing or improperly implemented tokens.
- **CSRF Tester**: A tool designed for manual CSRF testing. It allows you to create and send custom CSRF attack requests.
- **Burp Suite**: Use Burp's **Repeater** and **Intruder** for more advanced CSRF testing, especially if trying to brute-force token values or testing for weak randomness in tokens.

5. Use the SameSite Cookie Flag

- If the web application uses the `SameSite` cookie attribute for session cookies, ensure you test if the application properly blocks cross-site requests. `SameSite` flags can help mitigate CSRF by limiting how cookies are sent in cross-origin requests, but improper implementation can lead to bypasses.