**Web Application Security Testing Approach**

1. **Planning and reconnaissance**  
   The first stage involve: Defining the scope and goals of a test, including the systems to be addressed and the testing methods to be used.

Gathering intelligence (e.g., network and domain names, mail server) to better understand how a target works and its potential vulnerabilities.

1. **Scanning**The next step is to understand how the target application will respond to various intrusion attempts. This is typically done using:

**Static analysis** – Inspecting an application’s code to estimate the way it behaves while running. These tools can scan the entirety of the code in a single pass.

**Dynamic analysis** – Inspecting an application’s code in a running state. This is a more practical way of scanning, as it provides a real-time view into an application’s performance.

1. **Exploitation & Gaining Access**This stage uses web application attacks, such as cross-site scripting, SQL injection and backdoors, to uncover a target’s vulnerabilities. Testers then try and exploit these vulnerabilities, typically by escalating privileges, stealing data, intercepting traffic, etc., to understand the damage they can cause.
2. **Maintaining access**The goal of this stage is to see if the vulnerability can be used to achieve a persistent presence in the exploited system— long enough for a bad actor to gain in-depth access. The idea is to imitate advanced persistent threats, which often remain in a system for months in order to steal an organization’s most sensitive data.
3. **Analysis** The results of the penetration test are then compiled into a report detailing:

* **Planning and reconnaissance**

**Information Gathering:**

* 1. Manually explore the site
  2. Check for files that expose content, such as robots.txt, sitemap.xml, .DS\_Store Check the caches of major search engines for publicly accessible sites
  3. Check for differences in content based on User Agent (eg, Mobile sites, access as a Search engine Crawler)
  4. Perform Web Application Fingerprinting Identify
  5. Identify user roles
  6. Identify application entry points Identify client-side code
  7. Identify multiple versions/channels (e.g. web, mobile web, mobile app, web services)
* **Scanning**

1. Spider/crawl for missed or hidden content
2. Identify all hostnames and ports
3. Identify co- hosted and related applications
4. Identify technologies used
5. Identify third-party hosted content

* **Exploitation & Gaining Access**

**OWASP TOP 10 VULNERABILIT**

**Configuration Management/security misconfiguration:**

1. Check for commonly used application and administrative URLs
2. Check for old, backup and unreferenced files
3. Check HTTP methods supported and Cross Site Tracing (XST)
4. Test file extensions handling
5. Test for security HTTP headers (e.g. CSP, X-Frame-Options, HSTS)
6. Test for policies (e.g. Flash, Silverlight, robots)
7. Test for non-production data in live environment, and vice-versa
8. Check for sensitive data in client-side code (e.g. API keys, credentials)
9. Unpatched systems
10. Default/ out of the box account settings (i.e. usernames and passwords)
11. Unencrypted files
12. Old and out of date web applications
13. Unsecured devices
14. Web application and cloud misconfiguration
15. Insufficient firewall protection

How can I prevent security misconfigurations?

1. Developing a repeatable patching schedule
2. Keeping software up to date
3. Disabling default accounts
4. Encrypting data
5. Enforcing strong access controls
6. Provide admins with a repeatable process to avoid overlooking items
7. Set security settings in development frameworks to a secure value
8. Run security scanners and perform regular system audits

**Secure Transmission:**

1. Check SSL Version, Algorithms, Key length
2. Check for Digital Certificate Validity (Duration, Signature and CN)
3. Check credentials only delivered over HTTPS
4. Check that the login form is delivered over HTTPS
5. Check session tokens only delivered over HTTPS Check if HTTP Strict
6. Transport Security (HSTS) in use
7. Is any of this data stored in clear text long term, including backups of this data?
8. Is any of this data transmitted in clear text, internally or externally? Internet traffic is especially dangerous.
9. Are any old/weak cryptographic algorithms used?
10. Are weak crypto keys generated, or is proper key management or rotation missing?
11. Are any browser security directives or headers missing when sensitive data is provided by / sent to the browser?

**How To Prevent ‘Sensitive Data Exposure’?**

1. Considering the threats you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all sensitive data at rest and in transit in a manner that defends against these threats.
2. Don’t store sensitive data unnecessarily. Discard it as soon as possible. Data you don’t have can’t be stolen.
3. Ensure strong standard algorithms and strong keys are used, and proper key management is in place.
4. Ensure passwords are stored with an algorithm specifically designed for password protection,
5. Disable autocomplete on forms collecting sensitive data and disable caching for pages that contain sensitive data.
6. Consult Information Security Experts for detailed and thorough checks of all sensitive web applications.
7. Consider investing in DLP solutions or for Web Applications a WAF with custom rules (mask credit card numbers, SIN numbers) with targeted policies to prevent sensitive data exposure to clients
8. Classify data processed, stored or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
9. Apply controls as per the classification.
10. Don’t store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
11. Make sure to encrypt all sensitive data at rest.
12. Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
13. Encrypt all data in transit with secure protocols such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).
14. Disable caching for response that contain sensitive data.
15. Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt or PBKDF2.
16. Verify independently the effectiveness of configuration and settings.

**Authentication:**

1. Test for user enumeration
2. Test for authentication bypass
3. Test for bruteforce protection
4. Test password quality rules
5. Test remember me functionality
6. Test for autocomplete on password forms/input
7. Test password reset and/or recovery
8. Test password change process
9. Test CAPTCHA
10. Test multi factor authentication
11. Test for logout functionality presence
12. Test for cache management on HTTP (eg Pragma, Expires, Max-age) Test for default logins
13. Test for user-accessible authentication history
14. Test for out-of channel notification of account lockouts and successful password changes
15. Test for consistent authentication across applications with shared authentication schema / SSO

Remediation

1. Password length: Minimum password length should be at least eight (8) characters long. Combining this length with complexity makes a password difficult to guess using a brute force attack.
2. Password complexity: Passwords should be a combination of alphanumeric characters. Alphanumeric characters consist of letters, numbers, punctuation marks, mathematical and other conventional symbols.
3. Username/Password Enumeration: Authentication failure responses should not indicate which part of the authentication data was incorrect. For example, instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" for both. Error responses must be truly identical in both display and source code.
4. Protection against brute force login: Enforce account disabling after an established number of invalid login attempts (e.g., five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed.

**Session Management:**

1. Establish how session management is handled in the application (eg, tokens in cookies, token in URL)
2. Check session tokens for cookie flags (httpOnly and secure)
3. Check session cookie scope (path and domain)
4. Check session cookie duration (expires and max-age)
5. Check session termination after a maximum lifetime
6. Check session termination after relative timeout
7. Check session termination after logout
8. Test to see if users can have multiple simultaneous sessions
9. Test session cookies for randomness
10. Confirm that new session tokens are issued on login, role change and logout
11. Test for consistent session management across applications with shared session management
12. Test for session puzzling
13. Test for CSRF and clickjacking

Remediation

1. Credentials should be protected: User authentication credentials should be protected when stored using hashing or encryption.
2. Do not expose session ID in the URL: Session IDs should not be exposed in the URL (e.g., URL rewriting).
3. Session IDs should timeout: User sessions or authentication tokens should be properly invalidated during logout.
4. Recreate session IDs: Session IDs should be recreated after successful login.
5. Do not send credentials over unencrypted connections: Passwords, session IDs, and other credentials should not be sent over unencrypted connections.

**Authorization:**

1. Test for path traversal
2. Test for bypassing authorization schema
3. Test for vertical Access control problems (a.k.a. Privilege Escalation)
4. Test for horizontal Access control problems (between two users at the same privilege level) Test for missing authorization

**How to prevent Broken Authentication**

1. Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks.
2. Do not ship or deploy with any default credentials, particularly for admin users.
3. Implement weak-password checks, such as testing new or changed passwords against a list of the top 10000 worst passwords.
4. Align password length, complexity and rotation policies with NIST 800-63 B's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence based password policies.
5. Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
6. Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
7. Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.

**Error Handling and Logging**

1. Test for Auditable events, such as logins, failed logins, and high-value transactions are not logged.
2. Warnings and errors generate no, inadequate, or unclear log messages.
3. Test for Logs of applications and APIs are not monitored for suspicious activity.
4. Test for Logs are only stored locally.
5. Test for Appropriate alerting thresholds and response escalation processes are not in place or effective.
6. Test for Penetration testing and scans by DAST tools (such as OWASP ZAP) do not trigger alerts.
7. Test for The application is unable to detect, escalate, or alert for active attacks in real time or near real time.

**How to prevent a Insufficient Logging & Monitoring attack?**

1. Perform a baseline of logs needed for business which includes access logs, failed logins, suspicious or anomalous activities, network, endpoints, cloud etc.
2. Log formatted properly and context of logs is clearly understood
3. Have a centralized log management system where all logs are collected in one place like a SIEM tool integrated with real time reporting, heuristics and visualization tools
4. Synchronize time (UTC)
5. Secure the logs
6. Store the logs in accordance with the compliance and business requirements
7. Properly monitor user activity, anomalous behavior with automation and alerting
8. Log review should be closely monitored
9. Logs should not be deleted or modified
10. Integrate SIEM with SOC to improve threat detection and visibility
11. Legacy systems to cloud environments must be continuously monitored
12. Anomalous activity or any incident must be timely reported and action must be taken
13. Have an incident response plan following NIST 800-61 rev2 or later
14. Follow standards NIST 800-92, CIS control 6 and ISO27001
15. Perform pentesting and DAST tools to check to see where insufficient logging and monitoring has occurred

**Data Validation:**

1. Test for Reflected Cross Site Scripting Test for Stored Cross Site Scripting
2. Test for DOM based Cross Site Scripting
3. Test for Cross Site Flashing
4. Test for HTML Injection Test for SQL Injection
5. Test for LDAP Injection Test for ORM Injection
6. Test for XML Injection Test for XXE Injection
7. Test for SSI Injection
8. Test for XPath Injection
9. Test for XQuery Injection
10. Test for IMAP/SMTP Injection
11. Test for Code Injection
12. Test for Expression Language Injection
13. Test for Command Injection
14. Test for Overflow (Stack, Heap and Integer)
15. Test for incubated vulnerabilities Test for HTTP Splitting/Smuggling
16. Test for HTTP Verb Tampering
17. Test for Open Redirection
18. Test for Local File Inclusion
19. Test for Remote File Inclusion
20. Compare client-side and server-side validation rules
21. Test for NoSQL injection
22. Test for HTTP parameter pollution
23. Test for auto-binding
24. Test for Mass Assignment
25. Test for NULL/Invalid Session Cookie

**XSS prevention Techniques**

1. Blacklist filtering.
2. Whitelist filtering.
3. Contextual Encoding.
4. Input Validation.
5. Content Security Policy.

**SQL Injection prevention Techniques**

1. Validate User Inputs

2. Sanitize Data By Limiting Special Characters

3. Enforce Prepared Statements And Parameterization

4. Use Stored Procedures In The Database

5. Actively Manage Patches And Updates

6. Raise Virtual Or Physical Firewalls

7. Harden Your OS And Applications

8. Reduce Your Attack Surface

9. Establish Appropriate Privileges And Strict Access

10. Limit Read-Access

11. Encryption: Keep Your Secrets Secret

12. Deny Extended URLs

13. Don’t Divulge More Than Necessary In Error Messages

14. No Shared Databases Or User Accounts

15. Enforce Best Practices For Account And Password Policies

16. Continuous Monitoring Of SQL Statements

17. Perform Regular Auditing And Penetration Testing

18. Code Development & Buying Better Software

**Insecure Deserialization**

1. Object and data structure related attacks where the attacker modifies application logic or achieves arbitrary remote code execution if there are classes available to the application that can change behavior during or after deserialization.
2. Typical data tampering attacks such as access-control-related attacks where existing data structures are used but the content is changed.

Serialization may be used in applications for:

1. Remote- and inter-process communication (RPC/IPC)
2. Wire protocols, web services, message brokers
3. Caching/Persistence
4. Databases, cache servers, file systems
5. HTTP cookies, HTML form parameters, API authentication tokens

How to prevent Insecure Deserialization attacks?

1. You should never trust data when you deserialize it. Actions you can take to prevent insecure deserialization are:
2. Use data formats such as JSON or XML
3. Implement digital signatures on any serialized objects
4. Isolate an run code that deserializes in low privileged environments
5. Log deserialization exceptions and failures
6. Restrict or monitor network connectivity from containers or servers that deserialize
7. Use a Web Application Firewall

**DATA PROTECTION**

1. Use SSL everywhere
2. Disable HTTP access for all SSL enabled resources
3. Use the Strict Transport-Security header
4. Store user passwords using a strong, iterative, salted hash
5. Securely exchange encryption keys
6. Set up secure key management processes
7. Disable data caching using cache control headers and autocomplete
8. Limit the use and storage of sensitive data
9. Use valid SSL certificates from a reputable CA
10. Disable weak SSL ciphers on servers

**Denial of Service:**

1. Test for anti-automation
2. Test for account lockout
3. Test for HTTP protocol DoS
4. Test for SQL wildcard DoS

PREVENTING A DoS ATTACK

1.Work with your cloud and Internet service providers to implement service level agreements that include DoS defense provisions.

2.Ensure your system administrators are familiar with DoS protection service

3.Monitor network and systems

4.Install and configure firewalls and intrusion prevention systems.

5.Update and patch operating systems and applications

6.Use a website hosting service that emphasizes security.

7.Defend your network perimeter.

8.Plan for an attack.

**Business Logic:**

1. Test for feature misuse
2. Test for lack of non-repudiation
3. Test for trust relationships Test for integrity of data
4. Test segregation of duties

How to prevent business logic vulnerabilities

1. Make sure developers and testers understand the domain that the application serves
2. Avoid making implicit assumptions about user behavior or the behavior of other parts of the application
3. Maintain clear design documents and data flows for all transactions and workflows, noting any assumptions that are made at each stage.
4. Write code as clearly as possible. If it's difficult to understand what is supposed to happen, it will be difficult to spot any logic flaws. Ideally, well-written code shouldn't need documentation to understand it. In unavoidably complex cases, producing clear documentation is crucial to ensure that other developers and testers know what assumptions are being made and exactly what the expected behavior is.
5. Note any references to other code that uses each component. Think about any side-effects of these dependencies if a malicious party were to manipulate them in an unusual way.

**Cryptography:**

1. Check if data which should be encrypted is not
2. Check for wrong algorithms usage depending on context
3. Check for weak algorithms usage
4. Check for proper use of salting Check for randomness functions

**Risky Functionality - File Uploads:**

1. Test that file size limits, upload frequency and total file counts are defined and are enforced
2. Test that file contents match the defined file type
3. Test that all file uploads have Anti-Virus scanning in-place.
4. Test that unsafe filenames are sanitized
5. Test that uploaded files are not directly accessible within the web root
6. Test that uploaded files are not served on the same hostname/port
7. Test that files and other media are integrated with the authentication and authorization schemas

**Risky Functionality - Card Payment:**

1. Test for known vulnerabilities and configuration issues on Web Server and Web Application
2. Test for default or guessable password
3. Test for non-production data in live environment, and vice-versa
4. Test for Injection vulnerabilities
5. Test for Buffer Overflows
6. Test for Insecure Cryptographic Storage
7. Test for Insufficient Transport Layer Protection
8. Test for Improper Error Handling
9. Test for all vulnerabilities with a CVSS v2 score > 4.0
10. Test for Authentication and Authorization issues Test for CSRF

**HTML 5:**

1. Test Web Messaging
2. Test for Web Storage SQL injection
3. Check CORS implementation
4. Check Offline Web Application

**Top Pen testing Tools to use**

1. Burp Suite
2. Fortify
3. Nessus
4. Metasploit
5. Nikto
6. NMAP/ZenMap
7. ZED Attack Proxy (ZAP)Wireshark
8. Hashcat
9. John the Ripper
10. Hydra
11. Aircrack-ng
12. sqlmap
13. MobSF
14. ZED Attack Proxy (ZAP)
15. SonarQube
16. Cain & Abel
17. KALI
18. Social-Engineer Toolkit
19. Sqlninja