JWT Security Testing/Penetration Testing Checklist

No.	Vulnerability/Test	Description	Testing Steps	Expected Results	Security Recommendations
1	**Sensitive Information Disclosure in JWT Claims**	Sensitive data such as passwords, keys, or internal IPs are included in JWT payloads, which can be decoded client-side, leading to exposure of sensitive information.	1. Obtain a JWT token from the application. 2. Decode the JWT payload using tools like jwt.io or any Base64 decoder. 3. Inspect the claims for sensitive information like passwords, hashes, or internal data.	No sensitive information should be present in the JWT payload. Presence indicates a vulnerability.	- Do not include sensitive information in JWT claims. - Store sensitive data securely on the server side. - Include only necessary, non-sensitive information in the JWT payload.
2	**Signature Not Verified**	The server does not verify the JWT signature, allowing attackers to modify the token payload without invalidating the token, leading to unauthorized access.	1. Modify the JWT payload (e.g., change "admin": false to "admin": true) without altering the signature. 2. Use the modified token to access protected resources. 3. Observe if access is granted.	The server should reject tokens with invalid signatures. Acceptance indicates a vulnerability.	- Always verify JWT signatures on the server side. - Use trusted libraries that enforce signature verification. - Avoid disabling signature verification options.
3	**Algorithm None Vulnerability**	The JWT header alg is set to "none", disabling signature verification and allowing token tampering.	1. Modify the JWT header to set "alg": "none". 2. Remove the signature part of the JWT. 3. Use the modified token to access the application. 4. Observe if access is granted.	The server should reject tokens with "alg": "none". Acceptance indicates a vulnerability.	- Disallow the use of the "none" algorithm. - Explicitly specify and enforce acceptable algorithms. - Configure JWT libraries to reject tokens with "alg": "none".
4	**Weak HMAC Secret (Brute-force Attack)**	Secret (Brute-force Attack) Weak or commonly used secrets are used with HMAC algorithms, making them susceptible to brute-force attacks to discover the secret key.	1. Obtain a valid JWT token. 2. Use tools like Hashcat or John the Ripper with a wordlist to brute-force the HMAC secret. 3. If the secret is found, sign a new token with modified claims.	The secret should be strong and not discoverable via brute-force. Discovery indicates a vulnerability.	- Use strong, randomly generated secrets with high entropy. - Rotate secrets periodically. - Avoid default or easily guessable secrets.

5	**Algorithm Confusion Attack (RS256 to HS256)**	Changing the alg from RS256 to HS256 causes the server to use the public key as the HMAC secret, allowing attackers to forge tokens.	1. Change the JWT header "alg" from "RS256" to "HS256". 2. Use the public key as the HMAC secret to sign the token. 3. Use the modified token to access the application. 4. Observe if access is granted.	The server should not accept tokens with altered algorithms. Acceptance indicates a vulnerability.	- Enforce strict algorithm validation. - Do not mix symmetric and asymmetric algorithms. - Configure the server to accept only expected algorithms.
6	**Missing or Ignored exp Claim (Token Without Expiration)**	Tokens lack an expiration (exp) claim, or the server does not enforce token expiration, allowing indefinite token use.	1. Check if the token includes an exp claim. 2. Modify or remove the exp claim. 3. Use the token to access resources. 4. Observe if access is granted without expiration enforcement.	The server should enforce token expiration. Acceptance of tokens without exp indicates a vulnerability.	- Include an exp (expiration) claim in tokens. - Set appropriate token lifetimes. - Ensure the server validates the exp claim. - Implement token revocation mechanisms if necessary.
7	**Cross-Service Audience Claim Not Enforced**	Tokens intended for one service (aud claim) are accepted by another, potentially leading to unauthorized access across services.	1. Obtain a token for Service A with "aud": "ServiceA". 2. Use the token to access Service B. 3. Observe if access is granted without proper audience matching.	The server should reject tokens with incorrect aud claims. Acceptance indicates a vulnerability.	- Enforce strict aud claim validation. - Configure services to accept tokens only for their intended audience. - Use separate signing keys for different services if possible.
8	**Modifying Data Without Modifying Signature**	Testing if the server verifies the signature by altering the payload without updating the signature.	1. Modify the JWT payload (e.g., change user roles) without changing the signature. 2. Use the token to access resources. 3. Observe if access is granted.	The server should reject tokens with tampered payloads. Acceptance indicates a vulnerability.	- Ensure signature verification is enforced. - Reject tokens with invalid signatures. - Use libraries that enforce signature checks by default.
9	**Token Origin Exposure (Client-Side Generation)**	Tokens are generated on the client side, exposing the secret key or signing process to attackers.	1. Analyze client-side code for token generation logic. 2. Attempt to extract the secret key or reverse-engineer the signing process. 3. Use the key to forge tokens.	Tokens should be generated server-side only. Client-side generation indicates a vulnerability.	- Generate tokens exclusively on the server side. - Do not expose secret keys or signing logic to clients. - Securely store secrets and limit access.
10	**Excessive Token Lifetime**	Tokens have very long expiration times, increasing the risk window if compromised.	1. Check the exp claim for token lifetime. 2. Assess if the duration is appropriate.	Tokens should have reasonable lifetimes. Excessively long	- Set appropriate token lifetimes based on risk assessments. - Use short-lived tokens with refresh tokens if necessary.

			3. Attempt to use old tokens to access resources.	lifetimes indicate a vulnerability.	- Balance usability with security needs.
11	**Embedding New Public Key in Header (CVE-2018-0114)**	Attackers include a new public key in the JWT header, tricking the server into using it for signature verification.	1. Modify the JWT header to include a new jwk with an attacker-controlled public key. 2. Sign the token with the corresponding private key. 3. Use the token to access the application. 4. Observe if access is granted.	The server should not accept untrusted keys. Acceptance indicates a vulnerability.	- Do not trust keys supplied in tokens. - Maintain a set of trusted keys on the server. - Validate tokens only against known keys. - Ignore unrecognized key parameters in headers.
12	**JWKS Spoofing via jku Header**	Manipulating the jku header to point to attacker-controlled keys, allowing token forgery.	1. Modify the jku header to point to an attacker-controlled URL hosting malicious keys. 2. Sign the token with the corresponding private key. 3. Use the token to access the application. 4. Observe if access is granted.	The server should not fetch keys from untrusted sources. Acceptance indicates a vulnerability.	- Restrict jku URLs to trusted endpoints. - Use allowlists for JWKS URLs. - Validate TLS certificates when fetching JWKS. - Avoid dynamic key retrieval based on token headers.
13	**kid Parameter Injection and Directory Traversal**	Exploiting the kid (Key ID) parameter to perform path traversal, SQL injection, or command injection.	1. Modify the kid value to include path traversal sequences (e.g., "../../etc/passwd"). 2. Inject SQL commands if kid is used in database queries. 3. Inject OS commands if kid is used in system calls. 4. Observe server responses for errors or unauthorized access.	The server should sanitize kid values. Vulnerabilities indicate improper validation.	- Validate and sanitize kid inputs. - Avoid using untrusted data in file paths or commands. - Use parameterized queries and input validation.
14	**SSRF via x5u and jku Headers**	Using x5u and jku headers to perform Server-Side Request Forgery attacks.	1. Modify x5u or jku headers to point to internal or attacker-controlled URLs. 2. Monitor if the server makes requests to these URLs. 3. Use DNS logging to detect SSRF attempts.	The server should not make unauthorized outbound requests. SSRF indicates a vulnerability.	- Validate and restrict x5u and jku URLs. - Use allowlists for acceptable URLs. - Implement network egress controls. - Avoid fetching remote resources based on untrusted token headers.

15	**x5c Header Certificate Injection**	Including attacker-controlled certificates in the x5c header to bypass signature verification.	1. Create a self-signed certificate and include it in the x5c header. 2. Sign the token with the corresponding private key. 3. Use the token to access the application. 4. Observe if access is granted.	The server should validate certificates against trusted CAs. Acceptance indicates a vulnerability.	- Enforce strict certificate validation. - Accept certificates only from trusted sources. - Implement certificate pinning if appropriate. - Validate the certificate chain and revocation status.
16	**ES256 Nonce Reuse Leading to Private Key Compromise**	Reusing the same nonce (k value) in ECDSA signatures can lead to private key disclosure.	1. Collect multiple JWTs signed with ES256 using the same nonce. 2. Use cryptanalysis techniques to derive the private key. 3. Sign new tokens with the private key. 4. Use forged tokens to access the application.	Nonce reuse should not occur. Vulnerability exists if private key is recoverable.	- Use secure cryptographic libraries. - Ensure unique, random nonces for each signature. - Avoid manual implementation of cryptographic algorithms.
17	**Predictable or Reused jti Claim Leading to Replay Attacks**	The jti (JWT ID) claim is predictable or reused, allowing token replay or prediction.	1. Analyze jti values for patterns. 2. Attempt to reuse or predict jti values. 3. Replay tokens to see if they are accepted. 4. Observe if token reuse is possible.	Tokens should be unique and non-reusable. Vulnerabilities indicate improper handling.	- Use secure, random jti values. - Implement token replay protection on the server. - Maintain a store of used jti values to detect duplicates.
18	**Cross-Service Relay Attack (Improper Scope Enforcement)**	Tokens with broad scopes allow access to unauthorized services due to improper claim validation.	1. Use a token from one service to access another service. 2. Observe if access is granted without proper authorization. 3. Check if scope or role claims are enforced.	The server should enforce scope and permissions. Acceptance indicates a vulnerability.	- Enforce strict scope and permission checks. - Validate claims like aud, scope, and roles. - Implement least privilege principles.
19	**Failure to Enforce Time-Based Claims (exp, nbf, iat)**	The server ignores time-based claims, allowing tokens to be used outside their valid time window.	1. Modify exp, nbf, and iat claims to invalid values. 2. Use the token to access resources. 3. Observe if access is granted despite invalid claims.	The server should validate time-based claims. Acceptance indicates a vulnerability.	- Ensure the server checks all time-based claims. - Reject tokens with invalid or expired claims. - Synchronize server clocks.

20	**Use of Insecure or Deprecated Algorithms**	Using weak or deprecated algorithms in the alg header makes tokens vulnerable to attacks.	1. Check the alg field for insecure algorithms. 2. Attempt known attacks against these algorithms. 3. Observe if tokens are accepted.	The server should accept only strong algorithms. Acceptance of weak algorithms indicates a vulnerability.	- Use secure algorithms like RS256 or ES256. - Avoid deprecated algorithms. - Keep cryptographic libraries updated. - Enforce algorithm validation on the server.
21	**Missing or Invalid iat (Issued At) Claim**	Tokens lack the iat claim or it is set incorrectly, leading to potential security issues.	1. Check for the presence and correctness of the iat claim. 2. Modify iat to an invalid value. 3. Use the token to access resources. 4. Observe if access is granted.	The server should validate the iat claim. Acceptance indicates a vulnerability.	- Include the iat claim in tokens. - Validate the iat during verification. - Reject tokens with invalid iat values. - Ensure accurate server time synchronization.
22	**Insecure Client-Side Storage of JWTs**	JWTs are stored in localStorage or sessionStorage, making them accessible via JavaScript and vulnerable to XSS attacks.	1. Review client-side code for token storage locations. 2. Test for XSS vulnerabilities that could expose tokens. 3. Attempt to read tokens via injected scripts. 4. Observe if tokens can be exfiltrated.	Tokens should be securely stored. Exposure indicates a vulnerability.	- Store JWTs in HTTP-only cookies. - Implement Content Security Policy (CSP). - Prevent XSS vulnerabilities through input validation and output encoding.
23	**Lack of Token Revocation Mechanism**	No method to revoke tokens, so compromised tokens remain valid until expiration.	1. Log out and attempt to use the old token. 2. Observe if access is still granted. 3. Attempt to revoke a token and test if it's invalidated.	The server should invalidate revoked tokens. Acceptance indicates a vulnerability.	- Implement token revocation strategies. - Use short-lived tokens with refresh tokens. - Invalidate tokens upon logout or critical events. - Consider token blacklisting or introspection.
24	**JWT Used Without Proper HTTPS Protection**	Tokens are transmitted over unsecured channels (HTTP), exposing them to interception.	1. Monitor network traffic to see if tokens are sent over HTTP. 2. Attempt to intercept tokens using tools like Wireshark. 3. Observe if the application allows operation over HTTP.	Tokens should be transmitted over HTTPS only. HTTP transmission indicates a vulnerability.	- Enforce HTTPS for all communications. - Implement HSTS headers. - Do not allow fallback to HTTP. - Validate SSL/TLS configurations.
25	**Using JWTs for Session Management**	JWTs are used as session tokens without considering	1. Analyze how JWTs are used for sessions.	Session management should be secure and robust. Vulnerabilities	- Align JWT usage with session management best practices. - Consider using opaque tokens.

	Without Proper Security	statelessness and security implications.	2. Check for issues like lack of invalidation or improper refresh mechanisms. 3. Attempt session hijacking or replay attacks. 4. Observe if sessions can be compromised.	indicate improper implementation.	- Protect tokens during storage and transmission. - Implement proper session invalidation and renewal.
26	**Cross-Site Request Forgery (CSRF) in JWT Authentication**	JWTs stored in accessible locations are susceptible to CSRF attacks.	1. Craft a CSRF attack that triggers a request including the JWT. 2. Observe if the server processes the request. 3. Check for CSRF protections like tokens or SameSite cookies.	The server should prevent CSRF attacks. Vulnerabilities indicate lack of protections.	- Use CSRF tokens or anti-CSRF measures. - Store JWTs in HTTP-only, SameSite cookies. - Avoid storing tokens in accessible client-side storage. - Implement server-side CSRF defenses.
27	**Stored JWTs Vulnerable to XSS**	JWTs in client-side storage are vulnerable to theft via XSS attacks.	1. Identify XSS vulnerabilities in the application. 2. Inject scripts to read and exfiltrate JWTs. 3. Use stolen tokens to access the application. 4. Observe if access is granted.	Tokens should be protected from XSS. Vulnerabilities indicate improper storage.	- Prevent XSS through input validation and output encoding. - Store tokens in HTTP-only cookies. - Implement CSP to mitigate XSS risks.