**Verifying Remote Endpoint Authentication Enforcement**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that when the application provides user access to a remote service, proper server-side authentication (such as username and password validation, token-based authentication, or other secure mechanisms) is performed at the remote endpoint — not just on the client side. |
| **Pre-conditions** | - APK/IPA production build available- Device/emulator access- Burp Suite / mitmproxy proxy setup- Valid and invalid test user credentials |
| **Test Data** | - Valid username and password- Invalid or manipulated credentials (e.g. empty password, invalid token, tampered session cookie) |
| **Test Steps** | 1️⃣ Intercept all login or authentication requests using **Burp Suite** or **mitmproxy** while performing login or service access actions from the app.2️⃣ Observe whether credentials are being transmitted to a remote endpoint (API) for verification.3️⃣ Replace valid credentials or tokens with invalid or malformed values, replay the request, and check the API’s response.4️⃣ Confirm that the remote endpoint properly validates credentials/tokens and returns appropriate error messages (HTTP 401/403, error JSON).5️⃣ Decompile the APK/IPA using **MobSF**, **JADX**, or **Hopper** and review login or service access functions.• Ensure no purely local validation is happening without backend verification.6️⃣ Check for static or hardcoded credentials or session tokens within app code (security anti-pattern).7️⃣ Attempt offline login or remote service access with Wi-Fi/data disabled.• Confirm access is denied without server communication and validation. |
| **Expected Result** | - All authentication attempts go through a secure remote endpoint (API).- Server validates credentials or tokens and responds with appropriate success or error responses.- No purely local credential or token checks for remote services.- Offline or forged credentials/tokens result in access denial.- No hardcoded or static secrets present. |
| **Actual Result** | (To be filled after testing — e.g. App allowed login with empty password locally when offline, bypassed remote validation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if remote authentication bypassable or local-only validation detected |
| **Evidence** | Example findings:• API request absent during login attempt (pure local validation).• API accepted invalid or forged tokens.• App allowed login without online connectivity.• Hardcoded username/password pair found in app binary.• No error message or access restriction when invalid token replayed. |
| **Mitigation Recommendation** | - Enforce all authentication at the remote endpoint (API), even if client performs preliminary validation.- Always validate credentials and session tokens server-side.- Reject forged, malformed, or expired tokens/credentials with appropriate HTTP status codes (401, 403).- Avoid hardcoded credentials in app code or local validation-only login implementations.- Regularly test authentication workflows in mobile pentests and API security audits.- Include server-side authentication checks in your mobile app secure coding and design standards. |

**Verifying Secure Stateful Session Management with Random Session Identifiers**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that if stateful session management is implemented in the mobile application, the backend issues securely generated, random, high-entropy session identifiers upon successful authentication — and that these session tokens are used for subsequent client requests without resending user credentials. |
| **Pre-conditions** | - APK/IPA production build available- Device/emulator access- Burp Suite / mitmproxy proxy setup |
| **Test Data** | - Valid and invalid user credentials- Captured session tokens/identifiers |
| **Test Steps** | 1️⃣ Intercept the login request using **Burp Suite** or **mitmproxy**.• Verify whether a session identifier (session token, cookie, or bearer token) is issued upon successful login.2️⃣ Observe the authentication response.• Check for Set-Cookie header or JSON response containing a session ID/token.3️⃣ Capture the session identifier and analyze:• Confirm it’s a high-entropy, random-looking string (not sequential or predictable).• Estimate entropy using randomness estimation tools if needed.4️⃣ Perform authenticated operations while intercepting subsequent requests.• Confirm only the session token/identifier is sent with requests (via Cookie header, Authorization header, or request parameter).• Ensure no password or user credentials are resent after login.5️⃣ Tamper test:• Remove or replace the session identifier in subsequent requests — confirm server returns 401/403 errors.6️⃣ Replay old or forged session tokens and confirm they are rejected if invalid or expired.7️⃣ Test session logout or expiry — confirm session token is invalidated and no longer usable for access. |
| **Expected Result** | - A secure, random, unpredictable session identifier is issued upon authentication.- All subsequent authenticated requests use this session token only — no password resending.- Session identifiers are high-entropy and unique per session.- Invalid or missing tokens result in 401/403 errors.- Session identifiers are invalidated on logout or expiry. |
| **Actual Result** | (To be filled after testing — e.g. API sent password in every request alongside session token, token was sequentially numbered) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if weak, sequential, or reused tokens found or if credentials are resent on each request |
| **Evidence** | Example findings:• Set-Cookie: sessionid=12345 detected — sequential token.• Password resubmitted in each API call alongside token.• Replay of expired session token succeeded unexpectedly.• Logout endpoint did not invalidate session token. |
| **Mitigation Recommendation** | - Always issue random, high-entropy session tokens (UUIDv4, cryptographically secure random generators).- Never transmit passwords in subsequent authenticated requests after login.- Bind session tokens to user sessions and validate them on each request.- Invalidate tokens immediately upon logout or timeout.- Reject missing, expired, or tampered tokens with 401/403.- Review session token generation and validation in secure code reviews and API pentests.- Consider using JWT or OAuth2 for stateless/session-based security with appropriate expiration and validation controls. |

**Verifying Secure Signed Token-Based Authentication**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that if stateless, token-based authentication is used (e.g. JWTs), the server issues tokens that are properly signed using secure cryptographic algorithms, and that tokens are validated server-side on each request without relying on unsigned or weakly signed tokens. |
| **Pre-conditions** | - APK/IPA production build available- Device/emulator access- Burp Suite / mitmproxy proxy setup |
| **Test Data** | - Valid user credentials- Captured token values |
| **Test Steps** | 1️⃣ Intercept authentication requests and responses using **Burp Suite** or **mitmproxy** during login or token generation.2️⃣ Capture the issued token (commonly in Authorization: Bearer <token> header or in JSON response).3️⃣ If JWT-based (usually header.payload.signature format):• Decode token using [jwt.io](https://jwt.io) or local tool.• Review alg field in JWT header — confirm it uses a secure algorithm:✔ Acceptable: HS256, RS256, ES256❌ Weak/dangerous: none, HS256 with public secrets, RS256 with tampered public keys4️⃣ Test token signature verification:• Modify payload data (e.g. change user\_id, role) without adjusting the signature.• Replay request with the forged token — confirm server rejects it with 401/403.5️⃣ (If public key signing used — RS256/ES256):• Confirm server properly validates signature with its trusted private/public key pair.6️⃣ Review MobSF static scan or decompiled APK for signs of hardcoded token secrets or weak token validation logic client-side (token signature validation should occur server-side only). |
| **Expected Result** | - Token is signed using a secure cryptographic algorithm (e.g. HS256, RS256, ES256).- Server verifies token signature on every authenticated request.- Forged or tampered tokens are rejected.- No none algorithm or unsigned tokens in use.- No hardcoded secrets or token keys present in app code. |
| **Actual Result** | (To be filled after testing — e.g. JWT signed with alg=none accepted by server, or HS256 with hardcoded secret found in app) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if unsigned, weakly signed, or poorly validated tokens are accepted |
| **Evidence** | Example findings:• JWT issued with alg: none header.• Modified token payload with same signature accepted.• JWT signed using HS256 with hardcoded secret found via decompilation.• Replay attack with expired or forged token succeeded. |
| **Mitigation Recommendation** | - Use strong cryptographic algorithms for token signatures:✔ HMAC-SHA256+ (HS256)✔ RS256 (RSA with SHA-256)✔ ES256 (ECDSA with SHA-256)- Avoid none or unsigned tokens entirely.- Enforce strict signature validation server-side for every authenticated request.- Rotate token signing keys regularly.- Avoid hardcoding secrets in app binaries.- Test token integrity validation during pentests and code reviews.- Prefer public/private key signing for high-sensitivity applications. |

**Verifying Session Termination on User Logout**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that when the user logs out of the mobile application, the backend securely terminates the associated session by invalidating the session token or server-side session, preventing further authenticated requests with the same token. |
| **Pre-conditions** | - APK/IPA build available- Device/emulator access- Burp Suite / mitmproxy proxy setup |
| **Test Data** | - Valid user credentials- Active session token captured after login |
| **Test Steps** | 1️⃣ Intercept authentication requests using **Burp Suite** or **mitmproxy** during a normal login to capture a valid session token (Authorization header, Cookie, or custom header).2️⃣ Perform several authenticated API requests using the valid token to confirm access works.3️⃣ Log out via the app’s logout function.4️⃣ Reuse the previously captured session token to make the same authenticated API requests manually via proxy repeater.5️⃣ Confirm the server now rejects requests using the invalidated token (HTTP 401 Unauthorized or 403 Forbidden response).6️⃣ Check for token removal or nullification in client-side storage (local storage, SharedPreferences, or keychain).7️⃣ Confirm that after logout, no sensitive cached data or valid session tokens remain on the device.8️⃣ (Optional) If session revocation isn’t immediate, test with token expiration mechanisms or forced logout triggers (e.g. from another device). |
| **Expected Result** | - Upon logout, the server invalidates the session token.- Subsequent requests with the old token result in 401/403 Unauthorized errors.- The app clears all local session data (tokens, credentials, sensitive caches).- User is redirected to login screen after logout. |
| **Actual Result** | (To be filled after testing — e.g. Old token still valid after logout, API responded 200 OK to reused session token) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if tokens remain valid after logout or session termination is incomplete |
| **Evidence** | Example findings:• API still accepted requests with old session token after logout.• No logout API call observed — app only cleared local preferences.• Token persisted in local storage after logout.• Logout function navigated user to login screen but left session valid. |
| **Mitigation Recommendation** | - Implement proper server-side session/token invalidation upon logout.- Reject requests with old or invalidated tokens using HTTP 401/403.- Clear all locally stored tokens, session data, and sensitive caches on logout.- Implement token blacklisting or revocation mechanisms if using JWT or OAuth2.- Include session termination verification in mobile pentests, API tests, and secure code reviews.- Log session termination events for audit purposes. |

**Verifying Password Policy Enforcement at Remote Endpoint**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application enforces a strong password policy at the backend (remote endpoint), validating password strength server-side — not solely via client-side controls — during user registration, password changes, and password resets. |
| **Pre-conditions** | - APK/IPA build available- Device/emulator access- Burp Suite / mitmproxy proxy setup |
| **Test Data** | - Various test passwords (empty, short, common, weak, strong, malformed, previously used) |
| **Test Steps** | 1️⃣ Intercept password-based registration and password change API requests using **Burp Suite** / **mitmproxy**.2️⃣ Attempt to register a new account or change password using weak passwords:• Short passwords (e.g. 12345)• Common passwords (password, admin123)• Password without uppercase, lowercase, numbers, or special characters3️⃣ Observe API response:• Confirm the server returns an error (HTTP 400/422) with a clear validation message.4️⃣ Bypass client-side restrictions:• Modify intercepted password values to weak ones via proxy even if app UI blocks them.• Replay the modified request.• Confirm server rejects invalid passwords regardless of client validation.5️⃣ Test for server-side password reuse prevention (if policy requires).• Attempt to reset password using a previously used one.• Confirm server rejects reuse.6️⃣ Review server error messages:• Ensure no verbose error messages exposing validation logic or policy details unnecessarily. |
| **Expected Result** | - Password strength validated server-side during all relevant operations (registration, change, reset).- Weak, short, or previously used passwords rejected by the backend.- Bypassing client-side password restrictions via proxy fails at the backend.- API returns clear, safe error messages (e.g. “Password must be at least 8 characters with uppercase, lowercase, number, and symbol.”). |
| **Actual Result** | (To be filled after testing — e.g. Server accepted '12345' password when client-side check was bypassed) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if weak or client-only password validation detected |
| **Evidence** | Example findings:• API allowed weak password abc123 via proxy while client blocked it.• No server error returned on 5-character password.• Password reuse not enforced — allowed setting same password after reset.• No password length or complexity policy enforced server-side. |
| **Mitigation Recommendation** | - Enforce strict password policies on the remote server:✔ Minimum length (8-12 characters)✔ At least one uppercase, one lowercase, one number, one special character✔ No common or compromised passwords✔ Optional: block password reuse- Validate password policy server-side on registration, change, and reset endpoints.- Return consistent, clear, non-verbose error messages.- Never rely solely on client-side validation.- Regularly test password policies in mobile pentests and backend API audits. |

**Verifying Remote Endpoint Protection Against Excessive Credential Submissions**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the remote endpoint implements a robust protection mechanism (such as rate limiting, temporary account lockout, progressive delays, or CAPTCHA challenges) to block excessive failed credential submissions, mitigating brute force and credential stuffing attacks. |
| **Pre-conditions** | - APK/IPA production build available- Burp Suite / mitmproxy proxy setup- Test user accounts |
| **Test Data** | - Series of invalid credentials (wrong passwords, non-existing usernames) |
| **Test Steps** | 1️⃣ Intercept login API requests using **Burp Suite** or **mitmproxy**.2️⃣ Craft a series of login attempts for a single valid username with invalid passwords.• Start with 5–10 rapid attempts and observe responses.• Monitor HTTP status codes and response times.3️⃣ Check if the server enforces any of the following after repeated failures:✔ Temporary account lockout (e.g. after 5 failed attempts)✔ Progressive delays before accepting new login attempts✔ HTTP 429 (Too Many Requests) response✔ Error messages indicating rate limiting or lockout✔ CAPTCHA challenge after multiple failures (if implemented)4️⃣ Attempt credential stuffing by testing multiple usernames with a known password.• Confirm rate limiting applies per IP, per account, or globally.5️⃣ Review app’s response to server messages:• Ensure no verbose errors like “Incorrect password for user X” that could aid enumeration.6️⃣ (Optional) Check if backoff counters reset after a reasonable time (e.g. 15-30 minutes). |
| **Expected Result** | - After a defined number of failed login attempts, the server enforces protections:✔ Temporary lockout or delay✔ Rate limiting (HTTP 429)✔ Generic, consistent error messages✔ Prevention of brute-force attacks on credentials- No verbose errors leaking account existence |
| **Actual Result** | (To be filled after testing — e.g. API allowed unlimited login attempts with no lockout or rate limiting) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if no effective brute-force protections or rate limiting present |
| **Evidence** | Example findings:• 50+ login attempts allowed for testuser without delay or lockout.• No HTTP 429 or progressive delay implemented.• Different error messages for invalid usernames vs. invalid passwords (account enumeration risk). |
| **Mitigation Recommendation** | - Enforce account-based and/or IP-based rate limiting for login attempts.- Implement temporary account lockout or CAPTCHA challenges after a defined number of failures.- Use HTTP 429 (Too Many Requests) with appropriate Retry-After headers.- Apply generic, non-verbose error messages for failed authentication.- Regularly test brute-force protections during mobile and API pentests.- Log and monitor excessive authentication failures for alerting and incident response. |

**Verifying Session and Token Expiry at Remote Endpoint After Inactivity**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application’s backend enforces session invalidation after a predefined inactivity period, and that issued access tokens automatically expire and are rejected server-side once expired — preventing stale session reuse or unauthorized access. |
| **Pre-conditions** | - APK/IPA production build available- Device/emulator access- Burp Suite / mitmproxy proxy setup |
| **Test Data** | - Valid credentials and active session token |
| **Test Steps** | 1️⃣ Log in via the mobile app and capture the session identifier (session cookie, token, or Authorization header) using **Burp Suite** or **mitmproxy**.2️⃣ Record the session/token expiry value if explicitly provided (e.g., exp in JWT payload, max-age in Set-Cookie).3️⃣ Let the session/token remain idle for longer than the expected inactivity timeout or token expiry period (e.g. 15-30 minutes).4️⃣ Attempt an API request using the same previously captured session token.5️⃣ Confirm the server now rejects the request with a proper error code:✔ HTTP 401 Unauthorized or 403 Forbidden✔ Consistent session expired/invalid response6️⃣ If a JWT is used:• Decode it with [jwt.io](https://jwt.io)• Check the exp (expiry) claim exists and is reasonably short-lived (e.g. 15-60 minutes for access tokens).7️⃣ (Optional) Verify refresh token mechanisms (if present) to securely renew sessions without exposing original credentials. |
| **Expected Result** | - Session tokens and access tokens expire after a predefined inactivity period or lifetime.- Server rejects requests using expired or idle tokens.- 401/403 errors consistently returned for expired sessions.- JWTs have valid, reasonable exp claim and server honors expiry check on every request.- Optional: refresh token securely renews session where applicable. |
| **Actual Result** | (To be filled after testing — e.g. Session token remained valid after 60 minutes of inactivity, no expiry enforced server-side) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if tokens/sessions remain indefinitely valid or expiry not enforced server-side |
| **Evidence** | Example findings:• API accepted 45-minute-old token with no expiry check.• No exp claim present in JWT.• No 401/403 returned after token idle period.• Session cookies missing Max-Age or Expires attributes.• Refresh token used insecurely or never expired. |
| **Mitigation Recommendation** | - Set and enforce short, reasonable lifetimes for access tokens (e.g. 15-60 minutes).- Include exp claim in all stateless tokens like JWTs.- Implement inactivity-based session expiry for stateful sessions.- Reject expired sessions/tokens server-side — never rely on client enforcement.- Return HTTP 401/403 with generic error message on session expiry.- If using refresh tokens, ensure they expire after a longer period (e.g. 1 week) and securely revokeable.- Regularly test session management during API and mobile pentests. |

**Verifying Biometric Authentication Bound to Keystore/Keychain Access**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application’s biometric authentication is securely implemented by unlocking a protected keystore (Android) or keychain (iOS) entry, rather than simply returning a true/false result that could be bypassed via runtime manipulation. |
| **Pre-conditions** | - APK/IPA production build available- Device/emulator with biometric capability- Frida / Objection / JADX / Hopper |
| **Test Data** | - Valid biometric (registered fingerprint/FaceID)- Attempts to bypass biometric checks via method hooking |
| **Test Steps** | **Android:**1️⃣ Decompile APK using **MobSF**/**JADX**.• Search for biometric prompt calls (BiometricPrompt.authenticate()) or FingerprintManager.authenticate().• Confirm whether success triggers are tied to secure Android Keystore keys via KeyGenParameterSpec with setUserAuthenticationRequired(true).• Look for use of Cipher/Signature initialized with a keystore-backed key after biometric success.• Avoid insecure logic like:java<br>if (biometricSuccess) {<br> proceedWithPayment();<br>}<br>2️⃣ Install app on test device and perform biometric authentication.3️⃣ Use **Frida** or **Objection** to hook biometric callbacks — forcibly return true without actual biometric scan.4️⃣ Confirm whether app proceeds with sensitive actions or if access is controlled via unlocking a protected keystore item instead.5️⃣ Review whether sensitive keys (token decryption key, auth token, or salary info) remain locked unless biometric unlock succeeds through Keystore-backed cryptographic operation.**iOS:**1️⃣ Use **class-dump**/**Hopper** to review LAContext evaluatePolicy() implementations.• Check whether sensitive actions occur on simple success == YES condition, or if a SecKeychain/Keychain item is unlocked.2️⃣ Test runtime manipulation via Frida:• Hook evaluatePolicy and forcibly return YES.• Confirm whether sensitive actions still require unlocking Keychain-stored credentials.3️⃣ Review key storage implementation for kSecAttrAccessControl flags ensuring biometryCurrentSet or userPresence enforced. |
| **Expected Result** | - Biometric authentication unlocks a Keystore/Keychain-protected key or encrypted value.- App sensitive actions depend on access to that unlocked key.- Insecure true/false callback handling avoided.- Runtime bypass via method hooking prevents access without actual biometric success unlocking keystore. |
| **Actual Result** | (To be filled after testing — e.g. Frida hook on biometric callback returned true and app granted access without unlocking keystore item) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if biometric check is boolean-only without Keystore/Keychain protection |
| **Evidence** | Example findings:• if (biometricSuccess) proceed(); logic detected.• No Android Keystore key generation or KeyGenParameterSpec with setUserAuthenticationRequired(true) configured.• Frida hook bypassed biometric check and performed payment transaction.• No use of SecAccessControl or Keychain protection in iOS biometric workflows. |
| **Mitigation Recommendation** | - **Android:**• Use Android Keystore to generate secure keys bound to biometric authentication:```javaKeyGenParameterSpec.Builder builder = new KeyGenParameterSpec.Builder("keyAlias", PURPOSE\_ENCRYPT |

**Verifying Remote 2FA Implementation and Consistent Enforcement**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm whether the mobile application enforces a second factor of authentication (2FA) via the backend, and that 2FA requirements are consistently validated at the remote endpoint — preventing bypass attempts through client tampering, request manipulation, or incomplete 2FA flows. |
| **Pre-conditions** | - APK/IPA production build available- Burp Suite / mitmproxy proxy setup- 2FA enabled test accounts (if possible) |
| **Test Data** | - Valid username/password- Valid and invalid 2FA codes (OTP, push notification, or biometric triggers) |
| **Test Steps** | 1️⃣ Perform a standard login through the mobile app while intercepting requests via **Burp Suite** or **mitmproxy**.2️⃣ Check for a distinct 2FA verification API request after successful primary credential validation.• Confirm 2FA code or factor is transmitted to the remote endpoint for validation.3️⃣ Attempt to bypass the 2FA step:• Send a valid username/password pair but skip or remove the 2FA request step and proceed with accessing protected APIs using only the primary session token.• Confirm whether the backend blocks access or still enforces 2FA validation status.4️⃣ Submit invalid or manipulated 2FA codes and observe server response.• Confirm server rejects requests with incorrect or missing 2FA factors (HTTP 401/403, or clear JSON error response).5️⃣ Check whether 2FA is consistently enforced across login, password reset, and high-risk operations (fund transfer, salary updates).• Attempt such operations with and without 2FA step completed.6️⃣ (Optional) Decompile APK or IPA using **MobSF/JADX/Hopper** and review code for insecure local-only 2FA enforcement.• No final authorization or session flag setting should occur solely on the client. |
| **Expected Result** | - 2FA requirement enforced at the backend on all critical operations.- 2FA validation API present and effective.- Access to sensitive APIs requires both primary authentication and successful 2FA validation.- Client cannot bypass 2FA by skipping requests or altering app behavior.- Invalid 2FA codes cause secure, consistent error responses. |
| **Actual Result** | (To be filled after testing — e.g. Client permitted API access after login without completing 2FA step) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if 2FA can be bypassed via request manipulation or incomplete backend validation |
| **Evidence** | Example findings:• No 2FA validation API present after login.• Access token issued immediately after password verification, allowing full access.• Invalid 2FA codes accepted.• High-risk actions (salary transfer) permitted without completed 2FA flow.• Decompilation shows client-only 2FA checks with no backend enforcement. |
| **Mitigation Recommendation** | - Enforce all 2FA validation at the remote endpoint.- Require successful 2FA code verification before issuing full session tokens or enabling access to sensitive APIs.- Apply 2FA consistently across:✔ Logins✔ Password resets✔ High-risk transactions (salary change, fund transfer)- Avoid client-only 2FA enforcement or UI-based restrictions.- Return generic, safe error messages for failed 2FA attempts.- Log all 2FA-related events for auditing and anomaly detection.- Regularly test 2FA workflows in mobile pentests and API security reviews. |

**Verifying Step-Up Authentication for Sensitive Transactions**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that sensitive or high-risk transactions within the mobile application (like salary changes, fund transfers, password resets) require a **step-up authentication** (OTP, biometric, or password re-authentication) at the time of transaction, enforced by the remote backend. |
| **Pre-conditions** | - APK/IPA production build available- Burp Suite / mitmproxy proxy setup- Valid credentials and session |
| **Test Data** | - Valid user credentials- Session token- Valid and invalid 2FA codes / biometric trigger attempts |
| **Test Steps** | 1️⃣ Log in to the mobile app and perform standard authenticated actions while intercepting requests using **Burp Suite** / **mitmproxy**.2️⃣ Identify sensitive transactions (e.g., salary approval, fund transfer, user profile changes).3️⃣ Attempt to initiate these sensitive transactions without triggering step-up authentication (no additional OTP, biometric, or password prompt).4️⃣ Observe API requests — confirm whether a **step-up authentication value** (OTP code, biometric challenge result, or password re-entry) is expected and validated by the remote backend.5️⃣ Attempt to bypass the step-up by replaying API requests or omitting the step-up factor.• Confirm server denies transactions with missing or invalid step-up factor.6️⃣ Submit forged, reused, or invalid step-up credentials (expired OTP, bypassed biometric hook).• Confirm server responds with HTTP 401/403 or appropriate error.7️⃣ (Optional) Use **Frida** or **Objection** to hook biometric or OTP handling functions client-side and forcibly return true — verify backend enforcement still blocks unauthorized transactions. |
| **Expected Result** | - All sensitive transactions trigger mandatory step-up authentication.- Remote server verifies step-up factor validity.- Transactions without step-up factor are denied (401/403).- No client-only controls enforce step-up without server-side validation.- OTPs/biometric challenges have short validity and single-use protections. |
| **Actual Result** | (To be filled after testing — e.g. Salary update request accepted without OTP validation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if sensitive transactions can proceed without server-validated step-up authentication |
| **Evidence** | Example findings:• Fund transfer API accepted request without verifying OTP.• Salary details update processed without triggering biometric prompt.• OTP re-used or forged value accepted.• No 401/403 response when step-up factor omitted in API request.• Client-side enforced step-up controls bypassable via Frida. |
| **Mitigation Recommendation** | - Enforce step-up authentication on all high-risk transactions at the backend:✔ Transaction-specific OTP validation✔ Password re-authentication✔ Device biometrics unlocking protected keys- Ensure server denies sensitive actions without valid step-up factor.- Enforce OTP expiration and one-time-use policies.- Avoid client-only step-up logic — always validate factors remotely.- Regularly test transaction flows for step-up enforcement in mobile pentests and API security reviews.- Log all step-up events for anomaly detection and audit trail. |

**Verifying User Notifications and Session Management for Sensitive Account Activities**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application notifies users of sensitive account activities (such as new device login, password changes, sensitive transaction approvals) and provides a session management interface where users can view device/session details (IP address, location, timestamp) and revoke or block specific sessions/devices. |
| **Pre-conditions** | - APK/IPA production build available- Burp Suite / mitmproxy proxy setup- Multiple devices/accounts |
| **Test Data** | - Valid user credentials- Valid and invalid login/session attempts from different devices/locations |
| **Test Steps** | 1️⃣ Perform a new login from a second test device (or emulator) using valid credentials while intercepting requests using **Burp Suite** or **mitmproxy**.2️⃣ Observe whether the user receives a notification (email, push notification, or in-app alert) indicating new device access or sensitive account activity.3️⃣ Check the app UI for a **session management or device management screen**.• Confirm it lists all active devices/sessions for the account.• Validate that each entry shows contextual info: device type, IP address, location, and login timestamp.4️⃣ Attempt to perform sensitive activities (password reset, salary changes, 2FA setup) — confirm notifications are sent.5️⃣ Attempt to use proxy/VPN to test IP/location tracking accuracy in device list.6️⃣ Test the ability to revoke or block a specific session/device via the app.• After revoking a session, confirm further API requests from that session token receive a 401/403 Unauthorized response.7️⃣ Review API responses for session/device listing — check that no excessive sensitive data is leaked. |
| **Expected Result** | - User notified of new logins, sensitive activities, and password changes via email, push, or in-app alerts.- App includes a session/device management UI listing active sessions with contextual info (IP, device, location, timestamp).- User can revoke/block specific devices, terminating their sessions immediately.- Revoked sessions receive a 401/403 on further API requests.- No unauthorized or excessive data exposure via device management APIs. |
| **Actual Result** | (To be filled after testing — e.g. No notifications sent for new login, no session management interface provided) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if users are not notified of new device logins or cannot manage/revoke sessions |
| **Evidence** | Example findings:• No email/push/in-app notifications sent for new login.• No device/session list available in app UI.• IP address and location missing in session data.• Revoked sessions remained active after attempted termination.• Session list API response exposed excessive internal metadata. |
| **Mitigation Recommendation** | - Implement notifications (email, push, in-app) for new device logins, password changes, sensitive transaction approvals.- Provide a session management UI showing:✔ Device name/type✔ IP address✔ Geolocation (approx.)✔ Login timestamp- Allow users to revoke/block sessions, forcing remote token invalidation.- Enforce 401/403 for revoked session tokens immediately.- Sanitize session listing API responses.- Regularly test session management functionality and notification delivery during security assessments. |

**RBAC Model**

| **Role** | **Resource** | **Allowed Actions** | **Notes** |
| --- | --- | --- | --- |
| **Employee** | Own payslip | View, Download | Cannot access others' data |
| Own personal details | View, Update (except salary info) |  |
| Tax & benefits overview | View |  |
| Payment history | View |  |
| 2FA settings | Enroll, Update, Disable (self only) |  |
| Devices & sessions | List, Revoke (own sessions) |  |
|  | Support tickets | Create, View (own tickets) |  |
| **Payroll Manager** | All employee payslips | View, Generate, Approve | Restricted to assigned departments or company-wide if scoped |
| Payroll runs | Initiate, Approve, Cancel, View history |  |
| Payment disbursement reports | View |  |
| Employee salary adjustments | View, Propose changes | Requires approval by Finance Officer |
| Tax filings | Generate, Submit |  |
| 2FA management for own account | Enroll, Update, Disable |  |
| **Finance Officer** | Payment authorizations | Approve, Reject | Critical — separate from Payroll Manager |
| Tax filings | View, Submit |  |
| Company account balances | View |  |
| Audit logs | View |  |
| **HR Manager** | Employee profiles | View, Create, Update (except salary or tax info) |  |
| Employment status | Activate, Deactivate, Archive |  |
| Payslip comments or inquiries | View, Respond |  |
| **Auditor** | Payslips (read-only) | View | No generate/download permission |
| Payroll run summaries | View |  |
| Audit logs | View |  |
| **Administrator** | All system settings | Create, Update, Delete | Super-admin only |
| User accounts | Create, Disable, Reset password |  |
| Permissions and roles | Assign, Revoke |  |
| Audit logs | View |  |
| API keys & integrations | Manage |  |
| **Support Agent** | Employee profiles (limited fields) | View (read-only) | No salary/tax data |
| Support tickets | View, Respond |  |
| System status | View | No access to production data |

**Verifying Remote Endpoint Authorization Enforcement (Based on Payroll App RBAC Model)**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the payroll application’s remote backend consistently enforces its authorization model server-side for every role, ensuring users can only perform actions and access resources allowed by their assigned roles, even when client-side restrictions are bypassed. |
| **Pre-conditions** | - APK/IPA build available- API documentation or recon- Valid test accounts for each role:Employee, Payroll Manager, Finance Officer, HR Manager, Auditor, Admin, Support Agent- Burp Suite / mitmproxy setup |
| **Test Data** | - Valid API requests for permitted and non-permitted actions for each role- Modified API requests for privilege escalation attempts |
| **Test Steps** | 1️⃣ **For each test account and role**:• Log in and capture valid session tokens.• Intercept permitted API calls (e.g., Employee viewing their own payslip).• Replay API calls to permitted resources — confirm success.2️⃣ Attempt to access restricted resources/actions not allowed for that role:• Modify API requests to access other users’ records or admin actions.• Example: as Employee, attempt to view another employee’s payslip by changing user\_id in request URL or body.• Example: as Payroll Manager, attempt to approve tax filings (Finance Officer-only action).3️⃣ Send these manipulated requests and observe responses.4️⃣ Confirm backend responds with appropriate HTTP 403 Forbidden (or 401 Unauthorized if session invalid).5️⃣ Test role separation for all sensitive actions:• Salary changes, Payroll run approvals, Account management.• Check step-up 2FA where required.6️⃣ Attempt to enumerate system functions not visible in the UI via direct API calls.• Check whether backend authorization model blocks those too. |
| **Expected Result** | - Backend enforces role-based permissions strictly, regardless of client-side UI restrictions.- Unauthorized actions and access attempts return HTTP 403 Forbidden.- Privilege escalation attempts via modified API requests are blocked.- Role-based access control applies to every sensitive action and data object. |
| **Actual Result** | (To be filled during pentesting — e.g. Employee could access other employee payslip via API manipulation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **Critical** if privilege escalation or authorization bypass is detected |
| **Evidence** | Example findings:• Employee API request modified user\_id and accessed another employee’s salary details (403 not enforced).• Payroll Manager performed tax filings (Finance Officer-only action) via direct API call.• API endpoint accepted unauthorized account disable request from HR Manager.• Step-up 2FA bypassed on salary change approval via API. |
| **Mitigation Recommendation** | - Consistently enforce backend authorization model via middleware or policy engine for every API endpoint.- Validate role and resource ownership on each request server-side.- Use role-based decorators/middleware (e.g., @requires\_role('payroll\_manager')).- Reject unauthorized actions with HTTP 403 Forbidden.- Regularly test authorization enforcement via API pentests and code audits.- Log all authorization failures for anomaly detection. |