**Process to identify mobile application architecture and connected services**

| **Step** | **Action** | **Tools / Methods** |
| --- | --- | --- |
| **1** | Obtain the latest APK or IPA file of the mobile application. | App download via device or directly from dev team. |
| **2** | Perform **static analysis** to extract app metadata, manifest, certificates, hardcoded URLs, endpoints, and services. | **MobSF, jadx, apktool** |
| **3** | Run the app on a device/emulator with a **proxy (Burp Suite)** intercepting traffic to capture all external communications. | Burp Suite, mitmproxy |
| **4** | Log in and navigate through all app features while capturing requests and responses. | Use test accounts where possible. |
| **5** | List all **domains, IP addresses, ports, API paths, cloud services, SDKs, and third-party integrations** communicating with the app. | Review proxy logs and static analysis reports. |
| **6** | Map out the architectural components: mobile client, backend servers, third-party APIs, CDN, authentication services, payment gateways, analytics platforms etc. | Diagram or tabular mapping |
| **7** | Review **permissions, exported components, API endpoints, data storage, and cryptographic implementation** in the app. | MobSF report, manual review |
| **8** | Identify all **security concerns** for each connected component/service. | Based on OWASP MASVS/MSTG and test results |

**Sensitive Data Handling and identification in Mobile Application**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether all sensitive data elements used by the mobile application are clearly identified, categorized, and appropriately protected during storage, transmission, and processing in compliance with security standards. |
| **Pre-conditions** | - APK or IPA file of the mobile application available.- Static analysis tool (MobSF, jadx, apktool) and dynamic analysis setup (Burp Suite, emulator/device).- List of app features and expected data flows. |
| **Test Data** | - APK: com.example.app\_1.0.0.apk- Test user credentials and transaction data- Sample PII, payment, and credential data |
| **Test Steps** | 1. Review app documentation (if available) for declared sensitive data categories (PII, tokens, payment data, etc.).2. Perform static analysis to identify hardcoded sensitive strings or variables (password, token, secret, cardNumber, etc.).3. Intercept app traffic using a proxy to observe transmission of sensitive data.4. Review local storage mechanisms (SharedPreferences, SQLite, files) for sensitive data at rest.5. Validate how sensitive data is processed in memory and whether it is cleared after use.6. Check for logging of sensitive information in log files or system logs. |
| **Expected Result** | All sensitive data types are clearly identified and consistently protected through encryption, secure storage, and safe transmission mechanisms. No sensitive data should be exposed in logs, local storage, or insecure API requests. |
| **Actual Result** | (To be filled after testing — e.g. Sensitive payment tokens found stored unencrypted in SharedPreferences.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | (High / Medium / Low based on findings — typically **High** if sensitive data is exposed) |
| **Evidence** | Example from dynamic analysis (Burp Suite capture):**Request containing sensitive data:**http POST /api/payment HTTP/1.1 Host: api.example.org Content-Type: application/json { "card\_number": "4111111111111111", "expiry\_date": "12/26", "cvv": "123" } **Issue:** Payment data sent unencrypted in plaintext request or stored insecurely in app storage.**MobSF finding:** Hardcoded token API\_SECRET="supersecretapikey" in strings.xml |
| **Mitigation Recommendation** | - Identify all sensitive data categories: authentication tokens, PII, financial data, health data, device identifiers.- Avoid storing sensitive data locally unless absolutely necessary.- Encrypt sensitive data at rest using AES-256.- Use Android Keystore/iOS Keychain for credentials, tokens, and encryption keys.- Always transmit sensitive data over HTTPS/TLS with certificate pinning.- Avoid logging sensitive data in application and system logs.- Wipe sensitive data from memory immediately after use.- Document all sensitive data elements and apply data classification policies. |
| **Sensitive Data Categories in Mobile Applications** | **Personally Identifiable Information (PII):**- Full name, email address, phone number, home address, date of birth, government IDs (passport, national ID, SSN, Aadhaar)**Authentication Credentials:**- Passwords, PINs, security questions/answers, OAuth tokens, API keys, session identifiers, refresh tokens**Financial Information:**- Card numbers, expiry dates, CVVs, account numbers, UPI IDs, transaction amounts, payment history**Health Data:**- Medical records, prescriptions, biometric data, health metrics**Device Identifiers:**- IMEI, IMSI, device ID, advertising ID (GAID/IDFA), MAC addresses**Location Data:**- GPS coordinates, cell tower IDs, Wi-Fi names**Transaction Data:**- Order IDs, transaction IDs, cart details, in-app purchase receipts |

**Secure Local Storage in Mobile Applications**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that sensitive data stored locally by the mobile application is protected using secure storage mechanisms such as Android Keystore, iOS Keychain, encrypted databases, and not stored in insecure formats like plaintext files or unprotected SharedPreferences. |
| **Pre-conditions** | - APK/IPA file available- Static analysis tools (MobSF, jadx, apktool)- Rooted/emulated device for filesystem access- Dynamic analysis tools (adb shell, Frida, objection) |
| **Test Data** | - Test user credentials, tokens, transaction IDs, card numbers |
| **Test Steps** | 1. Install and run the app on an emulator or test device.2. Use static analysis to identify sensitive data variables.3. Browse app local storage directories (e.g. /data/data/package\_name/) for plaintext files, unprotected SharedPreferences, SQLite DB files, cache files.4. Check for sensitive data stored without encryption.5. If applicable, verify Keystore/Keychain usage.6. Review for hardcoded sensitive strings in code. |
| **Expected Result** | Sensitive data is stored securely using the Android Keystore/iOS Keychain, encrypted database storage, or securely encrypted files. No sensitive data should be stored in plaintext locally. |
| **Actual Result** | (To be filled post-test.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | (High / Medium / Low) |
| **Evidence** | Example:/data/data/com.example.app/shared\_prefs/credentials.xml contained:<string name="access\_token">eyJhbGc...</string> (in plaintext) |
| **Mitigation Recommendation** | - Store credentials, tokens, and sensitive data using Android Keystore or iOS Keychain.- Use AES-256 encryption for local database storage (e.g. SQLCipher).- Avoid storing sensitive data unless necessary.- Do not store passwords, card numbers, CVVs on device.- Secure cache and temp files and delete them after use.- Apply ProGuard/R8 or DexGuard to obfuscate code containing sensitive references. |

**Token managemnet and session handling in mobile application**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that session tokens, OAuth tokens, API keys, and other authentication credentials are securely generated, transmitted, stored, and revoked when necessary, ensuring session security and preventing unauthorized access. |
| **Pre-conditions** | - APK/IPA file available- Proxy setup (Burp Suite, mitmproxy)- Test user accounts |
| **Test Data** | - User credentials, access tokens, refresh tokens |
| **Test Steps** | 1. Log into the app and capture authentication tokens (JWT, OAuth, API tokens).2. Check token lifespans and claims via decoded JWT.3. Inspect token storage locations (SharedPreferences, SQLite, Keychain).4. Test for token reuse, replay, and invalidation post-logout.5. Attempt to use expired tokens.6. Review token transmission security (TLS enforcement). |
| **Expected Result** | Tokens should be securely generated, transmitted over HTTPS, stored in encrypted containers (Keystore/Keychain), have proper expiry, and be invalidated on logout or expiry. Replay attempts should be blocked. |
| **Actual Result** | (To be filled post-test.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | (High / Medium / Low) |
| **Evidence** | Example:Access token eyJhbGc... stored in plaintext in SharedPreferences.Logout failed to revoke token on server. |
| **Mitigation Recommendation** | - Use secure, random, non-predictable tokens (JWT/OAuth2).- Store tokens in Android Keystore/iOS Keychain only.- Set short lifetimes for access tokens and use refresh tokens.- Invalidate tokens on logout and on password reset.- Use HTTPS with certificate pinning for token transmission.- Validate token expiration and claims on every request.- Implement token replay detection and prevention. |

**Generic threat model for mobile application**

| **Threat Surface** | **Attack Vector** | **Threat (STRIDE)** | **Example Scenario** | **Countermeasure** |
| --- | --- | --- | --- | --- |
| **Mobile Client (APK/IPA)** | Reverse engineering, code injection, APK repackaging | Tampering, Elevation of Privilege | Attacker modifies APK to bypass payment validation | Code obfuscation (ProGuard/R8), root detection, app integrity checks |
|  | Insecure local storage (SharedPreferences, SQLite) | Information Disclosure | Sensitive tokens stored unencrypted locally | Use Android Keystore / iOS Keychain, encrypt data at rest |
|  | Debuggable build in production | Elevation of Privilege | Debuggable flag allows runtime introspection | Disable debuggable flag in release builds |
| **API Server / Backend** | Broken authentication / token reuse | Spoofing, Information Disclosure | Attacker reuses stolen token for unauthorized access | Short token lifespans, token revocation, device binding |
|  | IDOR (Insecure Direct Object Reference) | Elevation of Privilege | User accesses another user's data by changing user\_id in request | Enforce RBAC/ABAC, object ownership validation |
|  | Excessive request flooding (DoS) | Denial of Service | API is overwhelmed with automated requests | Rate limiting, throttling, WAF protections |
| **Network Communication** | Man-in-the-middle (MITM) via compromised Wi-Fi | Tampering, Information Disclosure | Interception of unencrypted API traffic | Enforce HTTPS/TLS 1.2+, certificate pinning |
| **Cloud Storage (S3/Blob)** | Public bucket exposure | Information Disclosure | Publicly accessible file uploads expose sensitive documents | Enforce private bucket policies, signed URLs |
| **Third-Party SDKs / APIs** | Excessive telemetry data collection | Information Disclosure | SDK leaks PII or location data to external services | Vet third-party SDK privacy practices, restrict telemetry scope |
| **Push Notifications** | Sensitive data in notification payloads | Information Disclosure | OTP sent in plaintext push notification | Avoid sending sensitive data in notifications |
| **Authentication Service** | Brute-force login attacks | Spoofing, Denial of Service | Automated password guessing | Strong password policy, account lockout, MFA |

**Threat model for Payroll web application**

| **Threat Surface** | **Attack Vector** | **Threat (STRIDE)** | **Example Scenario** | **Countermeasure** |
| --- | --- | --- | --- | --- |
| **Web Login Portal** | Brute-force, credential stuffing | Spoofing, Denial of Service | Attacker attempts password guessing at scale | Rate limiting, account lockout, MFA, CAPTCHA |
|  | Session token theft via XSS | Spoofing, Information Disclosure | Malicious script captures active session cookie | Content Security Policy (CSP), input sanitization, HttpOnly/Secure cookies |
| **Payroll Processing Backend** | SQL Injection via parameter manipulation | Tampering, Information Disclosure | Attacker accesses salary records using SQL injection | Input validation, parameterized queries, WAF |
|  | Business logic flaw in salary adjustments | Elevation of Privilege | Regular user manipulates form values to increase own salary | Server-side validation of business rules, audit logs |
|  | IDOR on employee payslips | Information Disclosure | User accesses another employee's payslip by changing ID param | Enforce object ownership checks at server side |
| **API Endpoints** | Lack of authorization checks | Elevation of Privilege | API allows unauthorized access to employee bank account details | Strict RBAC enforcement on all endpoints |
|  | API abuse via rate flooding | Denial of Service | Automated requests exhaust payroll processing resources | Implement API rate limiting, quotas, WAF |
| **Data Storage (DB / Cloud)** | Unencrypted salary/payment data | Information Disclosure | Database stores salaries and PII in plaintext | Encrypt sensitive data at rest (AES-256) |
|  | Publicly accessible backup files | Information Disclosure | Payroll database backup exposed on public cloud bucket | Secure access policies, no public backups, cloud bucket audit |
| **Payroll Notification Emails** | Spoofed emails with phishing attachments | Spoofing, Information Disclosure | Employee receives fake payroll statements | Email SPF, DKIM, DMARC configuration, phishing awareness training |
| **Employee Self-Service Portal** | No session timeout / idle logout | Elevation of Privilege | Abandoned session remains active, accessible to another person | Implement session timeouts and inactivity logout |

**Threat model for payroll Mobile application**

| **Threat Surface** | **Attack Vector** | **Threat (STRIDE)** | **Example Scenario** | **Recommended Countermeasure** |
| --- | --- | --- | --- | --- |
| **Mobile Client (Payroll App)** | Reverse engineering, code tampering | Tampering, Elevation of Privilege | Attacker modifies the APK to bypass salary approval workflows | Use code obfuscation (ProGuard/R8), app integrity checks, root/jailbreak detection |
|  | Insecure local storage (tokens, salary data) | Information Disclosure | Payroll records and auth tokens stored unencrypted in SharedPreferences | Use Android Keystore / iOS Keychain, encrypt sensitive local data, secure cache clearing |
|  | Debuggable build in production | Elevation of Privilege | App shipped with debuggable flag allows runtime introspection | Disable debuggable flag in release builds |
|  | Hardcoded API keys and tokens | Information Disclosure | Static analysis reveals hardcoded production API keys | Remove hardcoded keys; fetch securely from backend via authenticated calls |
| **API Server** | Missing or weak authentication/authorization checks | Spoofing, Elevation of Privilege | Attacker calls /getPayslip API using another employee’s ID | Enforce OAuth2, RBAC, validate object ownership server-side |
|  | IDOR (Insecure Direct Object Reference) | Information Disclosure, Privilege Escalation | User modifies employee ID to access another's payroll details | Enforce server-side ownership checks on all sensitive resources |
|  | Lack of rate limiting / DoS protection | Denial of Service | Flooding the payslip download API to exhaust server resources | Implement API rate limiting, throttling, WAF protections |
| **Authentication Service** | Brute-force password guessing | Spoofing, Denial of Service | Automated attack attempts to guess employee logins | Enforce account lockout, MFA, strong password policy, and CAPTCHA |
|  | Missing session management (no token expiration / logout) | Elevation of Privilege | Session token remains valid after logout or inactivity | Implement session timeouts, token revocation APIs, inactivity logout |
| **Network Communication** | MITM attack on insecure Wi-Fi | Information Disclosure, Tampering | Interception of salary details transmitted over unencrypted HTTP | Enforce HTTPS/TLS 1.2+, certificate pinning, disable cleartext traffic |
| **Cloud Storage (Payslip PDFs)** | Publicly accessible cloud bucket | Information Disclosure | Employee salary PDFs stored in publicly readable S3 bucket | Enforce private bucket policies, signed URLs with short expiry, cloud IAM controls |
| **Push Notifications** | Exposing sensitive info in notification payloads | Information Disclosure | Push notification reveals salary processed for employee | Avoid including sensitive content in notifications; use status-only messages |
| **Payroll Processing API** | Business logic flaw in salary adjustments | Elevation of Privilege, Tampering | App user manipulates salary adjustment API parameters via proxy | Enforce business rules on server-side, validate inputs, audit logs |
| **Third-Party SDKs (Analytics, Crash reporting)** | Excessive telemetry capturing PII | Information Disclosure | Analytics SDK captures employee name and salary in logs | Review and limit third-party SDK permissions, enforce privacy policy alignment |
| **Device Storage & Backups** | Unprotected local app backups | Information Disclosure | Unencrypted payroll data backed up to cloud or device backups | Disable local backups, enforce backup encryption and protected directories |
| **Payroll Email Notifications** | Spoofed or malicious payslip emails | Spoofing, Information Disclosure | Employee receives fake salary adjustment notification | Use SPF, DKIM, DMARC, anti-phishing user training, MFA on self-service actions |

**Centralized Security Control Implementation**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that core security controls — including authentication, authorization, input validation, logging, rate limiting, error handling, and sensitive data protection — are implemented centrally at the backend (middleware/API gateway) or via reusable libraries and not inconsistently across endpoints or app modules. |
| **Pre-conditions** | - APK/IPA file available- Access to API endpoint URLs through dynamic proxy (Burp Suite / mitmproxy)- Backend/API gateway architecture documentation (if possible)- Static analysis tools (MobSF, jadx, apktool)- Test user accounts |
| **Test Data** | - Multiple API requests across different app features (login, payments, profile updates, file uploads, etc.) |
| **Test Steps** | 1. Intercept and test multiple API endpoints for:• Consistent authentication enforcement (OAuth, JWT checks)• Centralized authorization (role/permission check middleware)• Rate limiting and throttling headers (429 Too Many Requests)• Uniform error response structures (generic error messages without stack traces)• Input validation consistency (type checks, length, format validation on server side)• Consistent logging of security events (failed logins, invalid inputs, transaction attempts)2. Review app code for consistent use of shared libraries/utilities for sensitive actions (e.g. crypto helpers, input validation functions)3. Check whether any security controls are missing, duplicated, or inconsistently applied at individual endpoints. |
| **Expected Result** | All security checks should be enforced by centralized components — such as API Gateway, middleware, or shared backend services — and consistently applied to every request or sensitive operation. No endpoint should bypass these centralized controls. |
| **Actual Result** | (To be filled post-test — e.g. Only login endpoint uses JWT check; other APIs rely on client token without server validation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High (if inconsistencies found) |
| **Evidence** | Example findings:- /api/login uses JWT validation, rate limiting- /api/update-profile lacks JWT validation, accepts unauthenticated requests- /api/salary-upload missing input size check, allows 200 MB file upload- Error message on /api/transfer leaks SQL stack trace: com.mysql.jdbc.exceptions... |
| **Mitigation Recommendation** | - Enforce authentication and authorization through a centralized API Gateway, middleware, or server interceptor.- Implement rate limiting and request validation at API Gateway/WAF level.- Use shared libraries for input validation, logging, encryption, and error handling across backend services.- Define uniform API response formats (error messages, status codes).- Regularly perform code reviews and architectural audits for control centralization.- Document security policies and enforce through CI/CD security gates. |

**Insecure Configuration of Mobile Application and Instant Apps**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Identify misconfigurations and insecure settings in mobile applications (including Android Instant Apps) that could weaken security, enable reverse engineering, or expose sensitive features or data. |
| **Pre-conditions** | - APK/IPA or Instant App Bundle available- MobSF, jadx, apktool installed- Device/emulator and proxy configured for dynamic testing |
| **Test Data** | - AndroidManifest.xml and Info.plist files- App build config files- Local storage files, logs, and certificates |
| **Test Steps** | 1. Perform static analysis on APK/IPA using MobSF and jadx:• Check for android:debuggable="true"• Check for allowBackup="true" (Android)• Review usesCleartextTraffic="true" in manifest• Look for exported="true" components without intent filters or permissions• Find hardcoded API keys and sensitive config in code or resources• Review WebView settings for setJavaScriptEnabled(true) without domain restrictions• Check for excessive permissions beyond necessary (e.g. READ\_SMS, WRITE\_EXTERNAL\_STORAGE)• Identify presence of unused or test Activities, Services, Broadcast Receivers2. Dynamic test to confirm if exposed or misconfigured features are exploitable:• Attempt starting exported Activities via adb shell:adb shell am start -n com.example/.DebugActivity• Try sending crafted Intents to exported services• Test for cleartext HTTP requests and unprotected files |
| **Expected Result** | - Production app should have debuggable and allowBackup disabled- No cleartext traffic allowed (usesCleartextTraffic="false")- Exported components protected with permissions or removed- WebView hardened, no hardcoded sensitive config, minimal permissions |
| **Actual Result** | (To be filled after testing — e.g. Found debuggable flag enabled, exported Activity without intent filter restrictions) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High (if actively exploitable) |
| **Evidence** | MobSF static analysis:- android:debuggable="true"- usesCleartextTraffic="true"- Exported Activity: com.example.app.DebugActivity- Excess permissions: android.permission.READ\_SMS- WebView enabled JavaScript with no domain whitelistDynamic test:adb shell am start -n com.example/.DebugActivity successfully launched unauthorized Activity |
| **Mitigation Recommendation** | - Disable android:debuggable and allowBackup in production builds- Set usesCleartextTraffic="false" and enforce HTTPS-only traffic via Network Security Config- Mark internal components exported="false" unless required and protect with permissions or intent filters- Harden WebView settings: setAllowFileAccess(false), domain whitelisting, disable JavaScript unless necessary- Remove hardcoded secrets and sensitive config from app code- Limit requested permissions to the strict minimum- Regularly perform mobile static/dynamic config audits and enforce secure build configurations in CI/CD pipelines |

**Cryptographic Key managmenet Policy - Generic**

| **Policy Area** | **Policy Statement** |
| --- | --- |
| **Key Types and Usage** | All cryptographic operations must use strong, industry-standard algorithms: AES-256 for symmetric encryption, RSA-2048 or ECDSA P-256+ for asymmetric encryption, and SHA-256 or better for hashing. Cryptographic keys shall be used exclusively for their intended purpose (e.g., encryption keys should not be used for message signing). |
| **Key Generation** | Keys must be securely generated using system-approved cryptographically secure random number generators (CSPRNG). In Android, use KeyGenerator with SecureRandom from AndroidKeyStore; on iOS, use SecRandomCopyBytes. |
| **Key Storage (Client-side)** | Symmetric keys or secrets must never be hardcoded in the mobile application or stored in plaintext. Secure storage mechanisms must be used:- **Android:** Android Keystore API- **iOS:** iOS KeychainKeys should be marked non-exportable and hardware-backed wherever possible. |
| **Key Storage (Server-side)** | Keys on backend systems must be stored encrypted in a secure key management service (KMS), such as AWS KMS, Azure Key Vault, or Google Cloud KMS. Master keys must be stored separately from data keys. |
| **Key Transmission** | Keys must never be transmitted in plaintext over any network. If keys must be exchanged, they should be protected using asymmetric encryption (e.g., RSA/ECDH key agreement) or secure key-wrapping mechanisms, over TLS 1.2+ with certificate pinning. |
| **Key Rotation** | All keys must be rotated on a regular basis:- Data encryption keys (DEKs): every 90 days or immediately after suspected compromise- Session keys: per session or transaction- Master keys: annually or per regulatory requirements |
| **Key Expiry & Retirement** | Expired, unused, or compromised keys must be immediately revoked and securely destroyed using secure deletion methods appropriate for the storage medium (i.e., zeroization for memory keys, deletion from secure hardware modules). |
| **Key Backup & Recovery** | All critical encryption keys stored on server-side systems must be securely backed up in encrypted form and stored in a separate, access-controlled, and tamper-resistant location. |
| **Access Control & Logging** | Access to cryptographic keys must be strictly limited to authorized processes and personnel. All key access events must be logged and regularly reviewed. |
| **Algorithm Deprecation** | Weak or deprecated algorithms (MD5, SHA-1, DES, RC4) are strictly prohibited. Transition plans must be in place for migrating to newer standards as required. |
| **Cryptographic API Usage** | Only platform-approved APIs (Android Keystore, iOS Keychain, BouncyCastle for backend Java apps) should be used for cryptographic operations. No custom cryptographic implementations are permitted. |

**Cryptographic Key managmenet Policy - Payroll Based**

| **Policy Area** | **Policy Statement** |
| --- | --- |
| **Key Types and Usage** | All payroll-related encryption operations must use AES-256-GCM for symmetric encryption and ECDSA P-384+ for digital signatures. No key derivation functions (KDF) weaker than PBKDF2-HMAC-SHA-256 with 100,000 iterations or Argon2 are allowed. Separate keys must be used for data at rest, data in transit, and message integrity checks. |
| **Key Generation** | Symmetric keys, key pairs, and signing keys must be generated using hardware-backed cryptographic modules (Trusted Execution Environment / Secure Enclave / HSMs) wherever possible. All random values must be generated using system CSPRNGs only. |
| **Key Storage (Client-side)** | No long-term storage of payroll-related encryption keys on the mobile client is permitted. Session keys, tokens, and short-lived encryption keys may be stored using:- **Android:** Android Keystore hardware-backed storage only- **iOS:** iOS Keychain with Secure Enclave protectionKeys must be marked non-exportable and session-only where possible. |
| **Key Storage (Server-side)** | All payroll data encryption keys must be managed via enterprise-grade KMS with hardware-backed storage. Keys used for PII and salary data encryption must be encrypted with master keys stored in HSMs. KMS audit logs must be enabled and reviewed weekly. |
| **Key Transmission** | No symmetric or master keys may be transmitted over networks. If key exchange is required (e.g., for ephemeral session keys), ECDH key agreement over mutually authenticated TLS 1.3 connections with client certificate validation and certificate pinning is mandatory. |
| **Key Rotation** | Key rotation schedule:- Payroll data encryption keys (DEKs): every 30 days- Session keys: per session- Token signing keys: every 7 days- Master keys: annually or sooner if required by compliance (PCI DSS/ISO 27001)Rotation must not impact data availability (use key versioning strategies). |
| **Key Expiry & Retirement** | Immediate revocation and destruction of keys upon employee termination (for those with access), suspected compromise, or role change affecting access rights. |
| **Key Backup & Recovery** | Only master keys are backed up in encrypted form. DEKs are re-derived or re-generated as needed. Backups must be encrypted with backup-specific keys stored in isolated, access-controlled HSMs. |
| **Access Control & Logging** | Strict access control enforced via RBAC/ABAC on all key access operations. Dual control for master key access. Comprehensive key usage logging integrated with enterprise SIEM. |
| **Algorithm Deprecation** | Prohibited: DES, 3DES, RC4, MD5, SHA-1, RSA < 2048 bits. Transition to NIST-approved algorithms per FIPS 140-3 mandate. |
| **Cryptographic API Usage** | Only platform-approved and third-party validated cryptographic libraries (Android Keystore, iOS CryptoKit, AWS KMS SDKs) may be used. No custom crypto implementations or experimental libraries allowed in production payroll apps. |
| **Regulatory Compliance** | All cryptographic controls must align with applicable standards (PCI DSS, GDPR, NIST SP 800-57, ISO 27001 Annex A.10). Key management policies and audit logs must be reviewed quarterly. |

**Mechanism for enforcing mobile application updates exists**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the mobile application includes a reliable mechanism to detect and enforce application updates (both mandatory and optional), preventing the use of outdated or vulnerable app versions in production environments. |
| **Pre-conditions** | - APK or IPA file of the mobile application available.- Test server or staging environment (if update enforcement relies on backend version checks).- Proxy tool (Burp Suite / mitmproxy) to inspect version check API calls.- Android/iOS device/emulator with outdated app version installed (if possible). |
| **Test Data** | - Test user credentials- Application version metadata (current and outdated versions)- Simulated old app version if feasible |
| **Test Steps** | 1. Install an older version of the app on a test device or emulator (if available).2. Launch the app and observe if it automatically checks for updates or allows login without update checks.3. Intercept initial API calls during app launch with Burp Suite / mitmproxy and look for a version check request (e.g. GET /api/check\_version or GET /version.json).4. Review server response to confirm whether it indicates:• Latest version available• Mandatory update flag (if implemented)• Optional update message5. Attempt to bypass the update check (e.g., block update URL, modify local config) and access app features.6. Validate whether access to critical features is blocked when an outdated app version is detected.7. Review static analysis (MobSF/jadx) for any hardcoded version enforcement logic. |
| **Expected Result** | - The app should reliably check for updates on launch or at login.- It should enforce mandatory updates if the installed version is below a critical minimum supported version.- The mechanism should prevent bypass via client-side manipulation.- App should display a clear, unskippable message or force-close behavior for mandatory updates. |
| **Actual Result** | (To be filled post-test — e.g. App allowed full access with outdated version, no update check performed) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High (if outdated, vulnerable versions remain functional) |
| **Evidence** | Example:Intercepted request:GET /api/check\_version HTTP/1.1 Host: api.example.comResponse:{"latest\_version":"3.0.0","mandatory\_update":true}**App Behavior:** Outdated app still allows login and money transfer features without update enforcement. |
| **Mitigation Recommendation** | - Implement a mandatory update mechanism triggered on app launch and/or login.- Use a backend-controlled API (/api/check\_version) to enforce version validation dynamically.- Set mandatory update flags and enforce app termination or restricted access for unsupported versions.- Avoid relying on client-side-only update checks.- Include update enforcement as part of CI/CD release management and mobile DevSecOps pipeline.- Log and monitor version check requests and app versions in use.- Prompt users with a blocking message or redirect to the app store for forced updates. |

**Test for minimum supprted OS Version**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify whether the mobile application enforces a minimum supported OS version requirement, preventing installation and execution on devices running unsupported or insecure OS versions. |
| **Pre-conditions** | - APK/IPA file available- Test device/emulator running an older OS version- Static analysis tools (MobSF, jadx)- Proxy tools (if backend version checks are involved) |
| **Test Data** | - Device running older Android/iOS versions (e.g., Android 7, iOS 11)- Test user credentials |
| **Test Steps** | 1. Install the app on a device or emulator running an unsupported OS version.2. Attempt to launch the app and observe behavior.3. Inspect AndroidManifest.xml for minSdkVersion value (Android).4. Review IPA Info.plist for MinimumOSVersion entry (iOS).5. Intercept app launch traffic to check for server-side OS version validation (if implemented).6. Verify whether critical functions are blocked on unsupported OS versions.7. Attempt to bypass or spoof OS version and observe behavior. |
| **Expected Result** | The app should refuse to install, or block execution on unsupported OS versions, displaying a clear error message or redirect to update the OS. |
| **Actual Result** | (To be filled post-test.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | Medium to High |
| **Evidence** | Example:• AndroidManifest.xml shows minSdkVersion=21 (Android 5.0).• Installed app on Android 6.0, app functions normally without warning.• No OS version validation implemented client-side or server-side. |
| **Mitigation Recommendation** | - Define appropriate minSdkVersion in AndroidManifest.xml and enforce in CI/CD.- Set MinimumOSVersion in iOS Info.plist.- Implement runtime OS version checks and display blocking message if unsupported.- Validate OS version in backend on login/session creation APIs.- Disable support for outdated OS versions in app store metadata and enforce security minimum baselines (Android 9+, iOS 14+). |

**Test for Jail break or root detection**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application detects and restricts execution on jailbroken or rooted devices, preventing execution in insecure environments. |
| **Pre-conditions** | - APK/IPA file available- Rooted/jailbroken device or emulator with root access- Frida, Magisk, or equivalent rooting tools |
| **Test Data** | - Rooted or jailbroken device- Test user credentials |
| **Test Steps** | 1. Install and run the app on a rooted Android device or jailbroken iPhone. 2. 2. Attempt to access app functionality. 3. Perform static analysis to detect root/jailbreak detection logic:• Android: search for checks like checkRootMethod1()• iOS: look for usage of APIs like stat("/Applications/Cydia.app") or fork() 4. Attempt to bypass detection using Frida/Magisk Hide.   5. Observe whether app terminates, disables functionality, or logs detection events. |
| **Expected Result** | The app should detect device compromise and either block access or disable sensitive features, showing a clear security warning message. |
| **Actual Result** | (To be filled post-test.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High |
| **Evidence** | Example:- Installed app on rooted Android 11 device, app allows full access without detection.- No root/jailbreak detection routines in MobSF or jadx static analysis. |
| **Mitigation Recommendation** | - Implement multi-method root/jailbreak detection (file existence checks, dangerous permission detection, system API anomalies).- Use hardware-backed SafetyNet (Android) or DeviceCheck (iOS).- Log and report detection events to backend.- Restrict sensitive operations or force logout when detected.- Obfuscate detection logic to hinder tampering/bypass via Frida. |

**Application Tamper detection mechanism**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application includes mechanisms to detect unauthorized modification, repackaging, or runtime hooking, preventing compromised app builds from executing. |
| **Pre-conditions** | - APK/IPA file available- APKTool, jadx for decompilation- Tampered APK build (modified app)- Frida, Magisk, or hooking frameworks |
| **Test Data** | - Original and modified APK files |
| **Test Steps** | 1. Modify the APK (e.g., change app icon, add custom code, or bypass license check logic).2. Sign the modified APK and reinstall.3. Launch app and observe behavior.4. Check for static tamper detection methods in code (e.g. SHA-256 hash checks, certificate signature validation).5. Perform runtime hooking with Frida and test if the app detects method interception or debug flags.6. Inspect logs or app crash behavior on tamper detection. |
| **Expected Result** | The app should detect unauthorized modifications and either terminate immediately or disable critical functionality. Tamper detection should be obfuscated to prevent easy bypass. |
| **Actual Result** | (To be filled post-test.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High |
| **Evidence** | Example:• Tampered APK installed successfully.• No checksum or signature verification triggered.• App allowed all operations without detecting modifications. |
| **Mitigation Recommendation** | - Implement certificate signature verification (compare current signature with expected signature hash).- Use checksum/hashing of critical classes and resource files at runtime.- Detect known hooking frameworks and debugging tools.- Integrate RASP (Runtime Application Self-Protection) libraries.- Obfuscate detection logic to prevent reverse engineering.- Invalidate tampered sessions and log incidents to backend. |

**Security addressed within all parts of SDLC**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that security is explicitly integrated, tested, and enforced at every stage of the mobile application software development lifecycle (SDLC) — from requirements to design, coding, testing, release, and maintenance — and that pentesting activities, code reviews, threat modeling, and secure release processes are properly documented, validated, and repeatable. |
| **Pre-conditions** | - Access to SDLC documentation and process artifacts- Interviews with project manager, dev leads, security team- Access to CI/CD pipelines, issue tracking, source code repo, static code analysis results- Access to security policies and audit logs |
| **Test Data** | - Secure coding guidelines- Security requirements documentation- Threat models (if available)- Static and dynamic analysis reports- CI/CD build logs- Past pentest reports and issue resolutions |
| **Test Steps** | **Phase 1: Requirements Gathering**✅ Confirm if explicit **security requirements** exist for:• Authentication (OAuth, MFA)• Data encryption (at rest, in transit)• Secure key management• Jailbreak/root detection• App update enforcement• Session management and timeout policies• Logging and monitoring requirements✅ Review if security acceptance criteria are documented in user stories (Jira/DevOps backlog).**Phase 2: Design & Architecture**✅ Check if **threat modeling** sessions are conducted per feature, covering:• Data flow diagrams (DFDs)• STRIDE classification of components (API, storage, mobile client, third-party SDKs)• Known mobile threats (Activity hijacking, insecure storage, reverse engineering, etc.)✅ Validate whether identified threats have corresponding security controls mapped.✅ Confirm secure design decisions (TLS 1.3 enforcement, SSL pinning, API Gateway security, RBAC/ABAC).**Phase 3: Development (Coding)**✅ Verify adoption of **secure coding guidelines**:• No hardcoded credentials, tokens, or keys• Secure random number generation (CSPRNG only)• Secure WebView configuration• Least privilege permission requests (e.g., no READ\_SMS, WRITE\_EXTERNAL\_STORAGE)✅ Confirm whether developers perform peer code reviews with secure code checklists covering:• Input validation (client & server)• Cryptography misuse (e.g., ECB mode, weak hashes)• API response handling• Authentication token management practices✅ Ensure **static application security testing (SAST)** tools are integrated into CI/CD (e.g., MobSF, SonarQube, Fortify, Checkmarx).**Phase 4: Testing**✅ Validate whether mobile-specific **dynamic application security testing (DAST)** is performed using Burp Suite, MobSF dynamic analysis, Frida for runtime tests.✅ Check for mobile-specific pentest activities:• Activity hijacking• Intent spoofing• Secure storage validation (Keystore/Keychain usage)• API authentication bypass tests• Rate limiting and DoS resilience• SSL pinning and bypass attempts✅ Confirm manual penetration testing is performed for every major release.✅ Review results from OWASP MASVS and MSTG mobile app security checklists.**Phase 5: Release & Deployment**✅ Confirm CI/CD pipeline enforces security checks before deployment:• SAST and DAST scans must pass• Dependency scanning (OWASP Dependency-Check, Snyk, JFrog Xray)• Hardcoded secret scanning (TruffleHog, GitLeaks)✅ Ensure only signed, obfuscated, non-debuggable release builds are uploaded to app stores.✅ Verify that mandatory **version and OS enforcement** is in place via APIs and manifest/plist settings.**Phase 6: Maintenance & Incident Response**✅ Confirm documented processes for regular app vulnerability management:• Regular security patch releases• Library/SDK update schedule (quarterly or upon CVE advisories)• Regular key rotation and certificate renewal policies✅ Verify active **mobile-specific security logging and monitoring**:• API abuse detection (rate limiting, anomaly detection)• Jailbreak/root detection logging• Failed login and suspicious activity logging✅ Check incident response runbooks exist for mobile-specific issues (compromised API keys, token leakage, app store tampering, SDK abuse). |
| **Expected Result** | Security controls, secure coding practices, threat models, pentesting procedures, and vulnerability management processes should be consistently applied and verifiable across all SDLC phases — with evidence (checklists, logs, reports) confirming coverage. No deployment or release should proceed without passing mandatory security gates. |
| **Actual Result** | (To be filled after assessment — e.g. No formal threat models or code review records; SAST partially integrated; no documented pentesting before releases) |
| **Status** | (Pass / Fail Detected / Partial Compliance) |
| **Severity** | High if major SDLC security phases are missing |
| **Evidence** | Example findings:• No documented security requirements in Jira user stories.• No threat model diagrams for mobile API interactions.• SAST tool not integrated into CI/CD pipeline.• Pentest reports exist but no documented fixes tracked in issue tracker.• Obsolete libraries flagged in OWASP Dependency-Check report, unresolved for 6 months. |
| **Mitigation Recommendation** | - Formalize a secure SDLC framework based on OWASP SAMM, BSIMM or ISO 27034.- Create mandatory security requirement checklists for requirements gathering phase.- Perform threat modeling for every release or major feature.- Integrate SAST, DAST, dependency scanning, and secret detection into CI/CD.- Maintain secure coding standards with enforced code reviews.- Include mobile-specific pentesting in pre-release test plans.- Track and fix all vulnerabilities in issue trackers, blocking release if unresolved.- Implement ongoing security patching, key rotation, and logging/monitoring controls.- Audit SDLC security activities quarterly and maintain compliance evidence. |

**3 RD party Library and Security Review Process**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the development team performs a formal security review of all third-party SDKs and libraries integrated into the mobile application, including licensing, vulnerability, and privacy impact assessments. |
| **Pre-conditions** | - Access to source code or MobSF dependency analysis reports- Dependency manifest files (Gradle, Podfile, package.json)- Access to project’s library approval workflow or policy documents |
| **Test Data** | - List of SDKs and libraries used in the app- Public CVE databases (NVD, Snyk)- Privacy policy of third-party services |
| **Test Steps** | 1. Use MobSF or apktool to extract and identify all SDKs/libraries integrated into the mobile app (static and dynamic libraries).2. Check for libraries with known vulnerabilities via:• OWASP Dependency-Check, Snyk CLI, GitHub Dependabot, JFrog Xray3. Confirm whether third-party SDKs (analytics, payment, social login) undergo a formal approval and security review before integration.4. Review licensing types (GPL, LGPL, MIT) for legal compliance.5. Assess SDK data privacy practices (telemetry, PII sharing, location tracking).6. Review for documented SDK security configurations (API keys obfuscated, HTTPS-only APIs, TLS pinning enforced). |
| **Expected Result** | Every third-party SDK should be vetted for security vulnerabilities, licensing, privacy compliance, and integration configuration. High-risk or unapproved libraries should be blocked or isolated. |
| **Actual Result** | (To be filled post-assessment.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if known vulnerable SDKs or unreviewed components are in use |
| **Evidence** | Example:• MobSF detected Firebase Analytics v17.0.0 with known CVE-2020-12345.• Snyk flagged Apache Commons v1.5.0 (critical RCE CVE-2019-1234).• No documented SDK privacy review process. |
| **Mitigation Recommendation** | - Implement formal third-party SDK security approval and review workflow.- Integrate Dependency-Check or Snyk scans into CI/CD pipeline.- Maintain up-to-date inventory of all SDKs and libraries.- Regularly update SDK versions and patch vulnerable components.- Conduct privacy assessments of SDK telemetry behavior and restrict PII exposure.- Avoid integrating SDKs with aggressive permissions or insecure data flows. |

**Secure CI/ CD Pipeine Design Validation [will be useful for future when integration with the Developers]**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the mobile application’s CI/CD pipeline enforces security controls, automated checks, and environment isolation for secure build, test, and deployment processes. |
| **Pre-conditions** | - Access to CI/CD configurations (Jenkins, GitHub Actions, GitLab CI, Azure DevOps)- Build scripts and deployment logs- Access to pipeline security policies |
| **Test Data** | - CI/CD config files (Jenkinsfile, build.gradle, YAML workflows)- SAST, DAST, Dependency scanning results |
| **Test Steps** | 1. Review pipeline stages for integration of security checks:• SAST (MobSF, SonarQube)• Dependency scanning (Snyk, OWASP Dependency-Check)• Hardcoded secret scanning (TruffleHog, GitLeaks)• Unit and regression test execution2. Confirm staging, testing, and production deployment environments are logically and physically isolated.3. Verify that only signed, release builds (non-debuggable, obfuscated) are deployed to stores.4. Check for tamper protection in CI/CD (build hash verification, signature checks).5. Review audit logs of CI/CD security scans and blocked deployments. |
| **Expected Result** | CI/CD pipeline must enforce all automated security controls, prevent insecure or untested builds from deployment, and maintain audit logs for security incidents or policy violations. |
| **Actual Result** | (To be filled post-assessment.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High |
| **Evidence** | Example:• No Dependency-Check integration in CI/CD.• Debuggable builds pushed to production environment.• No code signing validation step in pipeline. |
| **Mitigation Recommendation** | - Integrate SAST, DAST, dependency and secret scanning tools into CI/CD.- Enforce security gates that block build promotion if critical vulnerabilities exist.- Enforce signed and obfuscated release builds.- Isolate dev/test/staging/production environments.- Maintain centralized audit logs of pipeline activities.- Regularly test pipeline for bypass attempts or misconfigurations. |

**Mobile application incident detection and response procedures**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that incident detection, alerting, and response procedures are defined and operational for security events originating from the mobile app, its APIs, and backend infrastructure. |
| **Pre-conditions** | - Access to SIEM dashboards, alert definitions, and incident response playbooks- Application log configurations- Incident logs and ticketing system |
| **Test Data** | - Test security events (failed logins, token misuse, API abuse, jailbreak detection) |
| **Test Steps** | 1. Review SIEM or logging platform for coverage of mobile security events:• Jailbreak/root detection logs• Suspicious login attempts• API rate limit breaches• Token misuse or replay attempts• Error messages indicating business logic abuse2. Confirm whether mobile app security logs are centralized and reviewed.3. Check whether predefined alerts exist for high-risk events.4. Review incident response playbooks specific to mobile security events (token leakage, API key exposure, app tampering incidents).5. Test logging of simulated events (failed login bursts, tampered token usage). |
| **Expected Result** | The mobile application’s backend and security monitoring systems must log, detect, alert, and trigger incident response workflows for all high-severity security events within SLA timelines. |
| **Actual Result** | (To be filled post-assessment.) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High |
| **Evidence** | Example:• No logging of jailbreak detection events.• No alerts for repeated API abuse.• SIEM dashboard lacks mobile-specific incident categories. |
| **Mitigation Recommendation** | - Centralize all mobile security logs to SIEM (Splunk, ELK, Azure Sentinel).- Define high-priority alerts for jailbreak detection, API abuse, and token misuse.- Maintain up-to-date mobile incident response playbooks.- Conduct quarterly incident response drills for mobile attack scenarios.- Integrate security monitoring into CI/CD and release checklists.- Continuously improve detection rules based on emerging mobile threats. |

**Disclosure Policy [Future needed]**

Excellent — thank you for confirming! I’ll now provide a clean, polished **Responsible Disclosure Policy for a Payroll Mobile Application** drafted in DOCX-ready formatting (you can copy this directly into Word and apply your preferred style/template).

## 📄 ****Responsible Disclosure Policy****

**For: kPayroll Mobile Application**  
**Version:** 1.0  
**Effective Date:** 07 July 2025

### ****1. Purpose****

At **kPayroll**, the security and privacy of our customers’ personal, financial, and payroll data is our highest priority. We are committed to protecting our mobile application, APIs, and backend services by working closely with the security community to identify, assess, and resolve security vulnerabilities responsibly and promptly.

This Responsible Disclosure Policy outlines how external security researchers and ethical hackers can report potential vulnerabilities discovered in the **kPayroll mobile application or associated infrastructure**.

### ****2. Scope****

This policy covers the following digital assets and services:

**kPayroll Android and iOS mobile applications**

**kPayroll public APIs**

**Payroll processing backend infrastructure**

**Cloud services directly managed by kPayroll**

**Employee and partner web portals**

**Mobile push notification and payment services**

**Exclusions:**  
This policy does **not** cover:

Social engineering attacks against kPayroll staff or customers

Physical security vulnerabilities

Denial-of-Service (DoS/DDoS) testing without prior written consent

### ****3. Compliance and Standards Alignment****

This policy aligns with:

**ISO/IEC 29147:2018** — Vulnerability Disclosure

**ISO/IEC 30111:2019** — Vulnerability Handling

**OWASP Mobile Security Testing Guide (MSTG)**

**NIST SP 800-53 Rev 5** — System and Communications Protection

**GDPR Article 32** — Security of Processing

**GDPR Articles 33–34** — Data Breach Notification

### ****4. Reporting a Security Vulnerability****

If you discover a security vulnerability affecting our services, please report it to us responsibly using the process below:

📧 **Email:** [security@kpayroll.example.com](mailto:security@kpayroll.example.com)

Include:

Description of the vulnerability

Affected application, API, or component

Detailed steps to reproduce the issue (proof of concept, logs, screenshots)

Any potential impact assessment (optional)

**Public disclosure of vulnerabilities prior to remediation is strictly discouraged.**

### ****5. Vulnerability Handling and Response Process****

Upon receiving a valid report:

We will **acknowledge receipt within 3 business days**.

We will provide a preliminary assessment within **10 business days**.

Confirmed issues will be assigned a risk severity and remediation timeline:

Critical: 48 hours

High: 7 days

Medium: 30 days

Low: Next scheduled release

We will notify the reporter of the final resolution status.

Responsible researchers may be optionally credited in our public Security Hall of Fame.

### ****6. GDPR Compliance for Data-Related Vulnerabilities****

If the report involves exposure of **personal or payroll-related data** as defined by **GDPR Article 4**:

We will assess severity under **GDPR Article 32**.

If applicable, notify the supervisory authority within **72 hours**.

Notify affected users where required under **GDPR Article 34**.

Document all related actions under **GDPR Article 33(5)**.

We ask researchers to avoid unauthorized access, copying, or sharing of personal data. Any accidental access must be reported immediately.

### ****7. Safe Harbor Commitment****

Provided researchers:

Act in good faith

Follow this Responsible Disclosure Policy

Do not engage in data destruction, privacy violation, or service disruption

Provide us reasonable time to remediate issues

**kPayroll commits not to pursue legal action against you.** Your responsible research is welcomed and appreciated.

### ****8. Policy Review and Updates****

This policy is reviewed **annually** or after any major changes to:

Application architecture

Data protection regulations

Industry security standards

The latest version will always be available on our website at: [www.kpayroll.example.com/security](" \l ")

## 📑 ****Document Control****

| **Version** | **Date** | **Changes** | **Approved By** |
| --- | --- | --- | --- |
| 1.0 | 07 July 2025 | Initial policy creation | Security Officer |

## ✅ Next Step

Would you like me to prepare a **Word DOCX file layout with cover page, table of contents, document control footer, and styled headers** for this?

If yes — just confirm and I’ll package it for you properly 🚀

**Web application incident detection and response**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the web application’s infrastructure and backend services include clearly defined and operational security event detection, alerting, logging, and incident response procedures for handling web application security incidents. |
| **Pre-conditions** | - Access to SIEM or centralized logging platforms (Splunk, ELK, Azure Sentinel, etc.)- Access to web server logs, WAF logs, API gateway logs- Web application incident response runbooks or playbooks (if available)- Access to DevSecOps, SecOps, or platform team contacts |
| **Test Data** | - Simulated security events:• Failed login bursts• SQL injection probes• Brute-force attempts• Invalid session token usage• Suspicious user agent / geolocation anomaly- Known CVE test payloads for safe simulation |
| **Test Steps** | 1. Review centralized logging and monitoring configuration for:• Application access and error logs• WAF logs (AWS WAF, Cloudflare, etc.)• API logs (if API-driven)• Audit logs for admin and privilege escalation activities• Database access logs (if exposed via app)2. Confirm SIEM integration for log aggregation, correlation, and alerting.3. Verify that high-risk web incidents have detection rules:• SQL injection attempts• Excessive failed logins or credential stuffing patterns• Unauthorized session token replays• XSS payload detection in GET/POST requests• Elevated privilege usage by non-admins• Rapid API abuse / scraping attempts4. Review and validate incident response procedures:• Incident triage workflow with severity classification• Incident containment, notification, and mitigation steps• Evidence preservation policies (log retention, chain of custody)• Post-incident review and corrective action process5. Confirm runbooks exist for typical web incidents: • Data breach involving PII• Web app defacement• Credential stuffing attack• API key/token leakage |
| **Expected Result** | The web application environment should reliably detect and alert on high-risk web security events. Documented incident response procedures should exist for each major incident type, with clear response SLAs, escalation paths, and forensic data retention. |
| **Actual Result** | (To be filled post-assessment — e.g. No detection rules for API token misuse; no incident response runbooks for credential stuffing) |
| **Status** | (Pass / Fail Detected / Partial Compliance) |
| **Severity** | High if critical incidents go undetected |
| **Evidence** | Example findings:• No alerts for repeated failed login attempts on /login endpoint• WAF configured but no custom rules for known CVE payload patterns (SQLi, XSS)• No documented incident response process for API key leakage• No logs retained beyond 7 days for HTTP 5xx or failed logins |
| **Mitigation Recommendation** | - Integrate web server, WAF, and application logs with centralized SIEM.- Define and enforce alerting thresholds for high-risk web activities (failed logins, token misuse, SQLi/XSS attempts).- Establish clear incident response playbooks for:• Data breach• Credential stuffing• Application defacement• Unauthorized privilege escalation• API token leakage- Retain critical forensic logs (WAF, web server, API, DB) for at least **90 days** (as per ISO 27001 / PCI DSS recommendations).- Regularly review and test incident detection rules for new CVEs and OWASP Top 10 web application threats.- Conduct quarterly incident response drills simulating web application attacks. |

**Privacy compliance checklist**

| **✅** | **Control / Checkpoint** | **Regulation** | **Manual Verification Instructions** |
| --- | --- | --- | --- |
| ☐ | **Data Minimization** | GDPR Art. 5(1)(c) | 📌 Review each mobile feature and related APIs. Identify all personal data fields (e.g., name, salary, SSN).📌 Ask the business/data owner for a data dictionary or data use justification.📌 Confirm no unnecessary fields (e.g., gender, marital status) are collected without purpose. |
| ☐ | **Purpose Limitation** | GDPR Art. 5(1)(b) | 📌 Read the privacy notice.📌 Cross-check each data point with its described use case.📌 Confirm no data sharing with 3rd parties for unrelated processing (e.g., marketing). |
| ☐ | **Lawful Basis for Processing** | GDPR Art. 6 | 📌 Ask for the RoPA (Records of Processing Activities) and check payroll-related entries.📌 Confirm each data category (payroll, attendance, bank info) has a documented legal basis. |
| ☐ | **Privacy Notice Transparency** | GDPR Art. 13–14 | 📌 Open the mobile app and locate the privacy policy link (usually during onboarding or in settings).📌 Confirm inclusion of:• Data categories• Purpose• Legal basis• Retention• Contact info• Data rights |
| ☐ | **Consent Management (where needed)** | GDPR Art. 7 | 📌 Trigger optional features like analytics or biometric login.📌 Confirm opt-in consent is required and logged.📌 Review consent records (e.g., a consents table in DB or logs). |
| ☐ | **Data Subject Rights Functionality** | GDPR Art. 15–22 | 📌 Test account management screen for options like:• View my data• Edit profile• Request account deletion📌 Ask support if there's a DSAR (data subject access request) process and how it's logged. |
| ☐ | **Encryption In Transit and At Rest** | GDPR Art. 32, OWASP MASVS | 📌 Intercept app traffic using **Burp Suite** or **mitmproxy** — confirm TLS 1.2+ and no HTTP.📌 Check Android Keystore/iOS Keychain use via MobSF static scan or source code review.📌 Check backend DB config for AES-encrypted fields (e.g., salary data). |
| ☐ | **Secure Storage APIs (Keystore / Keychain)** | OWASP MSTG-STORAGE-2 | 📌 Decompile APK using MobSF or JADX.📌 Search for references to KeyStore, KeyChain, EncryptedSharedPreferences, etc.📌 Confirm sensitive items (JWT tokens, PINs) use these secure APIs. |
| ☐ | **Data Retention Policy** | GDPR Art. 5(1)(e) | 📌 Ask for the official retention schedule.📌 Confirm payroll records, tax data, and PII are deleted or archived after defined periods (e.g., 7 years).📌 Review backend job scheduler config or scripts that perform deletions. |
| ☐ | **Security Testing Before Releases** | GDPR Art. 32 | 📌 Ask for last 2 pentest or SAST reports.📌 Confirm coverage of mobile-specific risks (reverse engineering, insecure storage).📌 Validate issues were fixed (match Jira tickets or commit logs). |
| ☐ | **Incident Response & Breach Notification** | GDPR Art. 33–34 | 📌 Review incident response plan (should include mobile-specific threats).📌 Ask if response team performs simulations.📌 Confirm SLA for GDPR notification (72h) is defined and documented. |
| ☐ | **Access Control / RBAC** | GDPR Art. 32 | 📌 Log into the app as different roles (employee, manager).📌 Try accessing restricted features or endpoints.📌 Confirm unauthorized access is denied and logged. |
| ☐ | **Payroll Event Logging** | GDPR Art. 32 | 📌 Access log storage or logging dashboard (e.g., ELK, Splunk).📌 Search for:• Salary update logs• Admin login logs• Account deletion logs📌 Confirm logs are retained ≥ 90 days and protected from tampering. |
| ☐ | **No Hardcoded Secrets** | OWASP MSTG-CODE-7 | 📌 Use MobSF to analyze the APK.📌 Check for hardcoded API keys, auth tokens, or database credentials.📌 Look inside strings.xml, BuildConfig, and Retrofit/OkHttp client initializations. |
| ☐ | **OAuth2 + MFA for Authentication** | GDPR Recital 39 | 📌 Manually test login with invalid tokens and expired sessions.📌 Review app code and backend to confirm use of OAuth2 or OpenID Connect.📌 Confirm MFA is enforced for sensitive operations (e.g., salary release). |
| ☐ | **Third-Party SDK Review** | GDPR Art. 28 | 📌 Extract SDK list via MobSF or jadx.📌 Google each SDK and check privacy concerns, data collection behaviors.📌 Confirm contracts exist with 3rd-party vendors for data handling responsibilities. |
| ☐ | **Consent for Analytics / Telemetry** | GDPR Recital 32 | 📌 On first launch, check for analytics opt-in prompt.📌 Turn off consent and observe if telemetry stops.📌 Check if Firebase, Crashlytics, or other SDKs honor opt-out. |
| ☐ | **Data Residency Compliance** | GDPR Chapter V | 📌 Ask cloud provider or ops team for data center locations.📌 Confirm data is stored within EEA or with appropriate safeguards (e.g., SCCs, BCRs). |
| ☐ | **Privacy by Design / Default** | GDPR Art. 25 | 📌 Review app behavior:• Are sensitive features off by default?• Is least privilege enforced?📌 Ask for secure SDLC or threat modeling documentation. |
| ☐ | **DPIAs for High-Risk Features** | GDPR Art. 35 | 📌 Ask for recent DPIAs — focus on features like salary payouts, identity verification, or biometric login.📌 Confirm risks were assessed and mitigated before release. |
| ☐ | **OS Version / Update Enforcement** | OWASP MSTG-RESILIENCE | 📌 Install app on old Android/iOS versions and observe behavior.📌 Check manifest files for minSdkVersion and whether app enforces updates via backend. |
| ☐ | **RoPA (Records of Processing Activities)** | GDPR Art. 30 | 📌 Request a copy of the RoPA.📌 Confirm payroll-related data flows, processing purposes, categories, legal basis, and storage are documented. |