Mobile Application Key Management Security Validation

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the mobile application uses secure, platform-approved key management mechanisms for storing and managing cryptographic keys, and does not rely on insecure practices such as hardcoded symmetric keys, static key derivation, or weak local storage of encryption keys. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static analysis- Frida or Objection for dynamic runtime inspection- Test device or emulator |
| **Test Data** | - Application binary (APK/IPA)- Sample encrypted data (if accessible)- Any known keys or encryption parameters hardcoded or visible in code |
| **Test Steps** | 1. Perform **static code analysis** using MobSF and JADX:• Search for hardcoded keys in strings.xml, BuildConfig, or source files.• Identify uses of SecretKeySpec, Cipher.getInstance() and verify key initialization source.• Look for hardcoded passwords, base64-encoded keys, or shared secrets.2. Review encryption implementation logic:• Confirm AES keys are not hardcoded.• Check if Key Derivation Functions (PBKDF2, Argon2) are properly implemented with random salts and sufficient iterations.3. Verify usage of **platform Key Management APIs**:• Android: Android Keystore API with hardware-backed key storage.• iOS: Secure Enclave / Keychain APIs.4. Perform **dynamic analysis** using Frida or Objection:• Attempt to intercept key initialization at runtime.• Check if keys are loaded from hardcoded values, config files, or insecure storage.5. Review backend key management integration:• Confirm that sensitive master keys or data encryption keys are generated and stored on the server-side in secure KMS (e.g., AWS KMS, Azure Key Vault).• No key material should be transmitted to or stored in the client beyond ephemeral session keys if absolutely required. |
| **Expected Result** | - No hardcoded or statically defined symmetric keys present in app code or resources.- All keys used for encryption are securely stored and managed using platform-approved APIs (Keystore/Keychain).- Ephemeral keys (if any) are session-limited and securely exchanged using asymmetric encryption or derived securely.- Backend keys are stored in secure KMS, not exposed to the mobile client. |
| **Actual Result** | (To be filled after assessment — e.g. AES key hardcoded in CryptoUtils.java detected; no Keystore integration found) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if hardcoded keys or insecure storage found |
| **Evidence** | Example findings:• MobSF scan flagged SecretKeySpec("bXlTZWNyZXRLZXk=", "AES") hardcoded in CryptoUtils.java.• No Keystore API calls detected.• Frida trace confirmed runtime decryption key loaded from plaintext config.• No asymmetric key exchange used; symmetric key static per install. |
| **Mitigation Recommendation** | - Remove all hardcoded keys and static symmetric keys.- Use Android Keystore or iOS Keychain with hardware-backed, non-exportable key storage.- For sensitive data, store encryption keys only in backend KMS and perform cryptographic operations server-side.- If client-side encryption is necessary, generate per-session ephemeral keys exchanged securely over TLS with asymmetric encryption (e.g., ECDH).- Implement secure key derivation (PBKDF2 with ≥100,000 iterations, unique random salt).- Obfuscate key management code using ProGuard/R8 (Android) and anti-tampering techniques.- Conduct regular code reviews, static and dynamic analysis, and mobile-specific pentests focused on cryptography misuse. |

Verification of Cryptographic Primitives and Configuration

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application uses well-vetted, industry-standard cryptographic primitives (as per NIST, ISO/IEC, OWASP guidelines), and that common cryptographic configuration issues — such as weak key sizes, insecure modes, improper padding, hardcoded parameters, and custom implementations — are avoided. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static analysis- Frida or Objection for runtime inspection- Access to secure coding guidelines (if available) |
| **Test Data** | - APK/IPA binary- Encrypted data samples (if any)- App source code (if possible) |
| **Test Steps** | 1. Perform **static analysis** using MobSF and JADX:• Search for cryptographic API calls like Cipher.getInstance(), MessageDigest.getInstance(), Mac.getInstance(), KeyGenerator.• Identify algorithms and modes used (e.g., AES/ECB/PKCS5Padding).• Check key sizes for symmetric (≥256 bits for AES) and asymmetric keys (≥2048 bits for RSA, ≥256 bits for ECDSA).2. Look for deprecated or insecure algorithms:• DES, 3DES, RC4, MD5, SHA-1, RSA < 2048 bits.• ECB mode for block ciphers.• Static or null IVs in CBC/CTR modes.3. Review use of **platform-approved libraries**:• Android Keystore APIs, BouncyCastle (updated version), iOS CryptoKit.• Ensure no custom cryptographic implementations exist.4. Check **cryptographic configuration issues**:• Use of random, non-repeating IVs per encryption operation.• Key derivation functions use secure parameters (PBKDF2-HMAC-SHA-256 with ≥100,000 iterations, Argon2).• Secure random number generators (SecureRandom on Android, SecRandomCopyBytes on iOS).5. Perform **dynamic analysis** using Frida/Objection:• Hook into encryption calls and observe parameters (key sizes, IVs, modes).• Detect insecure runtime configurations or hardcoded keys. |
| **Expected Result** | - Only industry-approved cryptographic algorithms and secure configurations are used.- Deprecated algorithms, weak key sizes, insecure modes (e.g., ECB), or hardcoded cryptographic parameters do not exist.- Cryptographic operations use secure random IVs, proper padding, and hardware-backed storage where possible. |
| **Actual Result** | (To be filled after assessment — e.g. Detected AES/ECB/PKCS5Padding usage with 128-bit static key in CryptoUtils.java) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if insecure primitives or misconfigurations found |
| **Evidence** | Example findings:• Cipher.getInstance("AES/ECB/PKCS5Padding") found in CryptoUtils.java• 128-bit AES key hardcoded• SHA-1 hash used for password hashing• No random IV in AES-CBC implementation• No KDF — passwords directly used as keys |
| **Mitigation Recommendation** | - Replace insecure algorithms (e.g., AES/ECB, SHA-1) with modern alternatives (AES/GCM, SHA-256, Argon2).- Enforce minimum key sizes: AES-256, RSA-2048+, ECDSA P-256+.- Always use random, non-repeating IVs for each encryption operation.- Implement secure key derivation (PBKDF2-HMAC-SHA-256 with ≥100,000 iterations, or Argon2).- Use platform-approved cryptographic libraries only.- Remove custom crypto implementations and replace with standard libraries.- Regularly perform static/dynamic crypto reviews as part of mobile pentests.- Apply cryptographic misuse detection tooling (e.g., MobSF crypto analyzers). |

Secure Key Exchange Implementation Validation

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the mobile application implements secure, standards-based key exchange mechanisms for transmitting cryptographic keys, avoiding insecure static keys or weak protocols, and protecting against Man-in-the-Middle (MITM) attacks. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static analysis- Burp Suite/mitmproxy for TLS interception- Frida/Objection for dynamic hooking |
| **Test Data** | - Test session data (encrypted payloads, key exchange messages)- Known weak key exchange configurations (test payloads for ECDH, RSA key size tests) |
| **Test Steps** | 1. Perform **static analysis** of the app’s encryption logic:• Identify key exchange implementations (e.g. KeyAgreement.getInstance(), ECDH, RSA).• Check key sizes (RSA ≥ 2048 bits, ECDH P-256 or higher).2. Perform **dynamic analysis**:• Intercept key exchange traffic with Burp Suite (if possible).• Confirm that keys are exchanged over TLS 1.2+ with server certificate validation and optionally certificate pinning.3. Check if any **static symmetric keys** are exchanged or embedded within the app.4. Verify that **ephemeral keys** are generated per session (Perfect Forward Secrecy) using ECDHE.5. Use Frida to trace key generation methods at runtime, ensuring no hardcoded or static keys are used.6. Confirm asymmetric encryption (RSA or ECDH) is used for any sensitive data exchange over insecure channels (if unavoidable). |
| **Expected Result** | - Secure, ephemeral key exchange mechanism in place (e.g., ECDHE over TLS 1.2+).- No static symmetric key exchange.- No key material transmitted without secure wrapping (e.g., via asymmetric encryption).- All key exchanges protected by strong TLS with certificate validation and ideally pinning. |
| **Actual Result** | (To be filled post-test) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if static keys or insecure key exchange protocols detected |
| **Evidence** | Example findings:• Static symmetric key 0xA7B8... hardcoded in CryptoUtils.java.• No ephemeral ECDHE implemented.• TLS key exchange intercepted without pinning validation. |
| **Mitigation Recommendation** | - Implement ephemeral key exchange (ECDHE) over TLS 1.2+ with Perfect Forward Secrecy.- Enforce TLS certificate validation and pinning.- Remove all static symmetric keys from app code.- Use platform KeyStore / Keychain for key management.- Validate key sizes and algorithms per NIST/FIPS recommendations.- Conduct regular pentests and code reviews for cryptographic implementation flaws. |

**Validation of cryptography Primitives and their parameters for appropriate Use Cases**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application uses cryptographic primitives (encryption, hashing, signatures, key derivation) that are appropriate for their intended use cases and are configured with secure, industry-recommended parameters (algorithm, key size, mode, padding, IV generation, iteration count). |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static analysis- Frida/Objection for runtime inspection- Relevant documentation of cryptography design (if available) |
| **Test Data** | - App binary- Encrypted data samples, if retrievable- Simulated input to cryptographic functions |
| **Test Steps** | 1. Perform **static code analysis** using MobSF, JADX:• Identify cryptographic API calls (Cipher.getInstance(), Mac.getInstance(), MessageDigest.getInstance(), KeyAgreement, KeyGenerator).• Extract algorithm names, modes, key sizes, and parameters (e.g. AES/ECB/PKCS5Padding, RSA/1024, SHA-1).2. Map each primitive to its intended use case:• Symmetric encryption for data confidentiality (AES-256-GCM preferred)• Hashing for integrity (SHA-256 or higher)• Key derivation (PBKDF2-HMAC-SHA-256 with 100,000+ iterations, or Argon2)• Digital signatures (ECDSA P-256 or RSA-2048+)3. Verify appropriateness of each primitive:• Confirm no use of deprecated algorithms (DES, 3DES, RC4, MD5, SHA-1)• Confirm encryption modes avoid ECB; use GCM, CBC (with random IV)• Check key sizes meet modern standards (AES-256, RSA-2048+, ECDSA P-256+)• Verify random IV generation and secure padding schemes (PKCS7/PKCS5, or authenticated modes like GCM)4. Inspect **key derivation** implementations:• Check for use of random, per-user salts• Verify iteration counts (≥100,000 PBKDF2, or secure Argon2 config)5. Perform **dynamic analysis** with Frida/Objection:• Hook into cryptographic operations at runtime and confirm parameters being used.6. Review source or SDK documentation (if available) to confirm intended use cases and implementation details. |
| **Expected Result** | - All cryptographic primitives are suitable for their specific use case.- Algorithms, modes, key sizes, padding, and parameters align with OWASP MSTG, NIST, and ISO/IEC standards.- No weak or deprecated cryptographic primitives used.- Secure key derivation and random IV generation in place. |
| **Actual Result** | (To be filled post-test — e.g. AES used in ECB mode for salary data encryption; SHA-1 used for password hashing) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if weak primitives or insecure configurations found |
| **Evidence** | Example findings:• Cipher.getInstance("AES/ECB/PKCS5Padding") found in CryptoUtils.java• SHA-1 detected for password hashing with no salt• 128-bit AES key found, static per install• No IV generation for AES-CBC mode (IV = null) |
| **Mitigation Recommendation** | - Replace insecure algorithms (AES-ECB, SHA-1) with modern alternatives (AES-256-GCM, SHA-256 or higher).- Use random, non-repeating IVs per operation.- Enforce key sizes: AES-256, RSA-2048+, ECDSA P-256+.- Implement secure key derivation: PBKDF2-HMAC-SHA-256 (≥100,000 iterations) or Argon2 with random salt.- Use authenticated encryption modes (AES-GCM) for confidentiality + integrity.- Remove deprecated ciphers (DES, 3DES, RC4, MD5).- Conduct regular crypto configuration reviews during pentests and secure code audits. |

**Detection of Insecure and Cryptographic algorithm**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Identify any use of insecure, weak, or deprecated cryptographic algorithms and protocols in the mobile application and confirm that only modern, industry-approved algorithms are used for encryption, hashing, key exchange, and digital signatures. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static code analysis- Frida/Objection for dynamic runtime inspection- Documentation of approved cryptographic standards (e.g. NIST SP 800-131A, OWASP MSTG recommendations) |
| **Test Data** | - Application binary (APK/IPA)- API endpoints and encrypted data flows (if accessible) |
| **Test Steps** | 1. Perform **static analysis** using MobSF and JADX:• Search for cryptographic API calls such as Cipher.getInstance(), MessageDigest.getInstance(), Mac.getInstance(), Signature.getInstance().• Identify algorithms specified in these calls (e.g., "AES/ECB/PKCS5Padding", "SHA-1", "MD5").• Check TLS configurations (if present in manifest or code) for protocol versions (TLSv1, SSLv3).2. Cross-reference each detected algorithm with a list of **insecure and deprecated algorithms**:**Weak/Deprecated Algorithms:**• Symmetric: DES, 3DES, RC2, RC4, AES-ECB mode• Hashing: MD5, SHA-1• Asymmetric: RSA < 2048 bits, DSA < 2048 bits• Key Exchange: RSA key transport without ephemeral keys• TLS: SSLv3, TLS 1.0, TLS 1.13. Check for **custom or homegrown cryptography implementations** (classes named CustomCrypto, CryptoUtils, or custom cipher logic).4. Perform **dynamic analysis** with Frida/Objection:• Hook into cryptographic operations at runtime and log algorithms and parameters being used.• Intercept TLS handshakes and check protocol version negotiation and cipher suite selection.5. Confirm that only **approved modern algorithms** are used:• Symmetric: AES-256-GCM/CBC (with random IVs)• Hashing: SHA-256, SHA-384, SHA-512• Asymmetric: RSA-2048+, ECDSA P-256+• TLS: TLS 1.2+ (ideally TLS 1.3) |
| **Expected Result** | - No use of insecure or deprecated algorithms detected in code, libraries, API communication, or runtime operations.- Only strong, approved algorithms are used for each cryptographic operation type.- No custom cryptography implementations in production code. |
| **Actual Result** | (To be filled post-test — e.g. MD5 used for password hashing detected in CryptoUtils.java, TLS 1.0 connection allowed) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if weak or deprecated cryptographic primitives are in use |
| **Evidence** | Example findings:• MessageDigest.getInstance("MD5") found in CryptoUtils.java• Cipher.getInstance("AES/ECB/PKCS5Padding") detected• TLS handshake allowed TLS 1.0 protocol with weak cipher suite TLS\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA |
| **Mitigation Recommendation** | - Replace deprecated algorithms with modern, NIST-approved equivalents:• Use AES-256-GCM/CBC instead of DES, 3DES, or ECB.• Use SHA-256 or higher for all hashing operations.• Ensure RSA keys are ≥ 2048 bits, or use ECDSA P-256+.• Disable SSL, TLS 1.0, TLS 1.1 in transport configurations. Enforce TLS 1.2+ with strong cipher suites.- Remove any custom cryptography implementations.- Conduct regular static and dynamic cryptography reviews in pentests and secure code audits.- Apply mobile-specific cryptography misuse detection tools like MobSF and code review plugins. |

**Detection of Cryptographic keys re-use across multiple purposes**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application securely generates and manages cryptographic keys for specific, distinct purposes, and does not reuse the same key for multiple operations (e.g., using one key for both data encryption and message authentication), which would weaken the cryptographic strength and violate key separation principles. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static analysis- Frida/Objection for dynamic runtime inspection- Documentation of cryptographic key management (if available) |
| **Test Data** | - Application binary- Sample encrypted data (if possible)- Application-generated keys (if interceptable in memory or via logs) |
| **Test Steps** | 1. Perform **static code analysis** with MobSF and JADX:• Identify key generation or retrieval methods (KeyGenerator, SecretKeySpec, KeyStore.getKey()).• Track each key variable through the code and check its use in multiple contexts (e.g., both encryption and MAC, or for different data types).2. Inspect **cryptographic API calls** and verify that different keys are generated or retrieved for distinct purposes:• Separate keys for data-at-rest encryption, token signing, MAC generation, and session key exchanges.3. Look for hardcoded or constant keys that appear multiple times across modules or classes.4. Perform **dynamic analysis** with Frida/Objection:• Hook into key usage functions.• Trace key values and see if the same key is used for multiple operations (e.g., AES encryption and HMAC authentication, or both user data and salary data encryption).5. Review whether keys are **properly labeled, separated, and versioned** in secure storage (Keystore/Keychain) or backend KMS if applicable. |
| **Expected Result** | - Each key should be used exclusively for one defined purpose.- No cryptographic key is reused across unrelated operations.- Keys for encryption, signing, and authentication are logically and physically separated.- Key reuse for multiple functions is strictly avoided and prohibited in code and configuration. |
| **Actual Result** | (To be filled post-assessment — e.g. Static AES key used for both data encryption and token signing in CryptoUtils.java) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if key reuse for multiple purposes is found |
| **Evidence** | Example findings:• aesKey variable used for both data encryption (Cipher.getInstance("AES/CBC/PKCS5Padding")) and for HMAC generation (Mac.getInstance("HmacSHA256")).• Single master key hardcoded and reused for encrypting both PII data and salary details.• No key labeling or separation in Android Keystore entries. |
| **Mitigation Recommendation** | - Enforce strict key separation by generating and assigning different keys for each cryptographic purpose:• Separate encryption keys for different data domains (e.g., salary vs. identity data).• Distinct keys for data encryption, token signing, and message authentication.- Use hardware-backed Keystore/Keychain or server-side KMS with labels identifying key purpose and scope.- Remove any static, hardcoded master keys reused across functions.- Update secure coding standards to explicitly forbid cryptographic key reuse.- Conduct static/dynamic crypto reviews during secure code audits and mobile app pentests to detect key reuse patterns. |

**Validation of secure random number Generator**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that all security-sensitive random numbers (used for keys, IVs, nonces, salts, tokens) in the mobile application are generated using a sufficiently secure, cryptographically strong random number generator (CSPRNG), and not weak or predictable RNGs like java.util.Random or insecure third-party functions. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static analysis- Frida/Objection for dynamic runtime hooking |
| **Test Data** | - Application binary- Randomly generated keys, tokens, or values (if observable at runtime) |
| **Test Steps** | 1. Perform **static code analysis** with MobSF, JADX:• Search for random number generation API calls in app code: SecureRandom, Random, Math.random(), or any third-party RNG classes.2. Verify that **only CSPRNG APIs** are used for security-critical random values:• **Android:** java.security.SecureRandom• **iOS:** SecRandomCopyBytes• Backend: NIST SP 800-90A compliant DRBG or equivalent3. Identify usage contexts of random values:• Symmetric key generation• IV or nonce generation• Session or authentication token generation• Password salts• Cryptographic nonces4. Flag any use of weak RNGs in these contexts:• java.util.Random, Math.random(), UUID.randomUUID() for keys or crypto-related values.5. Perform **dynamic analysis** with Frida/Objection:• Hook into random number generation functions at runtime.• Test whether token/session key values generated across multiple app launches follow a predictable or static pattern.• Observe key, IV, or nonce generation logic and confirm SecureRandom/DRBG use. |
| **Expected Result** | - All security-critical random values generated using CSPRNG APIs.- No use of weak or insecure RNGs (Random, Math.random()) for any cryptographic or security-related purpose.- Random values are unpredictable across sessions and follow secure implementation practices. |
| **Actual Result** | (To be filled post-test — e.g. java.util.Random() detected in TokenGenerator.java for session token generation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if weak or predictable RNG found in cryptographic or security-critical use |
| **Evidence** | Example findings:• Random random = new Random(); used in TokenGenerator.java for session token generation• Math.random() detected in CryptoUtils.java for AES IV generation• Frida trace confirmed SecureRandom not invoked for token generation |
| **Mitigation Recommendation** | - Replace all instances of weak RNGs with **SecureRandom** (Android/Java) or **SecRandomCopyBytes** (iOS).- For keys, IVs, nonces, tokens, and salts, use only CSPRNGs that are hardware-backed where possible.- Enforce random value generation best practices:• AES IVs: random, unique per encryption• Password salts: unique, random per user• Session tokens: unpredictable, generated via SecureRandom- Add static code analysis checks for weak RNG usage.- Regularly conduct pentests and secure code reviews focused on RNG misuse. |

**Data Storage and Privacy**

**Validation of secure system credential storage for sensitive Data**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application uses system-provided secure credential storage mechanisms (Android Keystore, iOS Keychain) for storing sensitive data such as personal identifiable information (PII), authentication credentials, cryptographic keys, and session tokens, and avoids insecure local storage (SharedPreferences, NSUserDefaults, local files, SQLite) for such data. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static code analysis- Frida/Objection for dynamic runtime inspection |
| **Test Data** | - Sensitive app data (tokens, keys, user credentials, PII entries)- Application binary |
| **Test Steps** | 1. Perform **static code analysis** using MobSF and JADX:• Search for data storage API calls like SharedPreferences, FileOutputStream, SQLiteDatabase, NSUserDefaults, NSUserDefaults.standardUserDefaults().• Identify if sensitive data is stored through these mechanisms.2. Confirm use of **secure credential storage APIs**:• **Android:** AndroidKeyStore, EncryptedSharedPreferences, KeyProperties, Cipher, KeyGenerator.• **iOS:** Keychain Services, SecItemAdd, SecItemCopyMatching.3. Check for **encryption wrapping before storage** when system credential stores are not used (e.g., encrypting local files or databases with hardware-backed keys).4. Perform **dynamic analysis** with Frida/Objection:• Intercept calls to storage APIs.• Inspect runtime storage locations (e.g., /data/data/[package]/shared\_prefs/, /data/data/[package]/databases/, or NSUserDefaults files).• Check for presence of cleartext or weakly protected sensitive data.5. Review **Keystore/Keychain entries** via Frida:• Confirm that cryptographic keys, tokens, or passwords are stored in the platform-provided secure credential storage mechanisms. |
| **Expected Result** | - All sensitive data (credentials, PII, cryptographic keys) securely stored using system-provided secure storage mechanisms.- No cleartext or weakly encrypted sensitive data stored in app sandbox storage (SharedPreferences, files, SQLite, NSUserDefaults).- Cryptographic keys stored in non-exportable, hardware-backed system storage. |
| **Actual Result** | (To be filled post-test — e.g. JWT tokens stored in cleartext in SharedPreferences detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data is stored insecurely |
| **Evidence** | Example findings:• SharedPreferences.Editor.putString("token", jwtToken) in AuthManager.java• Password stored in NSUserDefaults without Keychain use• Private AES key stored in cleartext in SQLite database table config\_keys |
| **Mitigation Recommendation** | - Store all credentials, cryptographic keys, and sensitive tokens in:• **Android:** Android Keystore (hardware-backed, non-exportable)• **iOS:** iOS Keychain- Encrypt any local data at rest using platform-managed keys via Android Keystore or iOS Keychain.- Remove any storage of sensitive data in SharedPreferences, NSUserDefaults, or local files without protection.- Conduct secure code reviews and regular mobile pentests to verify storage hygiene.- Implement obfuscation and anti-tampering controls to protect storage API calls. |

**Validation of local storage practices for sensitive data**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application does not store sensitive data (PII, credentials, tokens, cryptographic keys) in insecure local storage locations such as external storage, shared storage directories, SD cards, public file directories, or application caches, and only uses secure system credential storage facilities (Android Keystore / iOS Keychain) or secure app container storage with encryption where necessary. |
| **Pre-conditions** | - APK/IPA file available- Test device or emulator with file system access- MobSF, JADX, apktool for static analysis- Frida/Objection for dynamic inspection |
| **Test Data** | - Sensitive data generated by the application (login tokens, session cookies, encrypted data, PII samples) |
| **Test Steps** | 1. Perform **static code analysis** with MobSF and JADX:• Search for data storage API calls such as FileOutputStream, openFileOutput, SharedPreferences, SQLiteDatabase, getExternalStorageDirectory, getExternalFilesDir, Environment.getExternalStorageDirectory().• Identify if sensitive data is written to external or shared storage locations.2. Check file paths referenced in code:• Ensure no sensitive data written to /sdcard/, /storage/emulated/0/, public directories like Pictures, Downloads, Documents or app cache directories.3. Perform **dynamic analysis**:• Use Frida/Objection to monitor file write operations at runtime.• Browse application directories via ADB shell:Android: /data/data/[package\_name]/, /sdcard/, /storage/iOS: App sandbox, cache, Documents directories.• Confirm no sensitive data written outside the app container or system credential storage.4. Verify that any local data (if necessary) is securely encrypted using hardware-backed Keystore / Keychain keys. |
| **Expected Result** | - No sensitive data (credentials, tokens, PII, keys) stored in external, public, shared, or unprotected local storage directories.- All sensitive data stored securely within system credential storage (Keystore / Keychain) or within the application’s private sandbox with encryption and proper access control. |
| **Actual Result** | (To be filled post-test — e.g. JWT tokens written to /sdcard/app\_data/token.txt detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data is stored insecurely |
| **Evidence** | Example findings:• FileOutputStream("/sdcard/app\_data/token.txt") storing access tokens• Encrypted salary records saved in app cache folder without encryption• Frida intercept showing token file write operation to external storage |
| **Mitigation Recommendation** | - Prohibit storing any sensitive data (PII, credentials, tokens, keys) in external storage or app cache directories.- Store all sensitive data in system credential storage:• **Android:** Android Keystore• **iOS:** iOS Keychain- If local storage is unavoidable (e.g., offline data caching), encrypt data at rest using AES-256-GCM with keys from the Keystore / Keychain.- Remove any sensitive debug logs or temporary files.- Implement regular mobile security code reviews and pentests focused on storage hygiene. |

**Validation that no sensitive data is written to application logs**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application does not log sensitive data — including PII, credentials, session tokens, cryptographic keys, or security event data — to application logs (Logcat for Android, Console logs for iOS), which could be accessed by attackers on compromised devices or via log aggregation services. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static analysis- Device/emulator with ADB log access (Android) / Xcode device logs (iOS)- Frida/Objection for dynamic runtime monitoring |
| **Test Data** | - App-generated sensitive data (login tokens, personal data, encrypted strings, session cookies) |
| **Test Steps** | 1. Perform **static code analysis** with MobSF and JADX:• Search for log statements: Log.d(), Log.i(), Log.e(), Log.v(), System.out.println(), NSLog().• Identify any logging of variables or function returns that may contain sensitive data (e.g., tokens, passwords, user identifiers, encrypted values).2. Examine **log messages and error handlers**:• Check if exceptions or crypto failures log stack traces or sensitive parameters.3. Perform **dynamic analysis**:• Run the app on an emulator/device.• Use adb logcat (Android) or Xcode device logs (iOS) to monitor log output.• Perform app actions involving sensitive data (login, payment, profile update).• Confirm no sensitive values appear in logs.4. Hook logging functions via **Frida/Objection**:• Intercept log statements at runtime.• Detect if sensitive variables are passed to logging methods. |
| **Expected Result** | - No sensitive data (tokens, PII, keys, passwords) logged to console or device logs.- Log messages are generic, sanitized, and contain no confidential business or personal data.- No sensitive data present in crash logs, analytics logs, or error traces. |
| **Actual Result** | (To be filled |

**Validation that no sensitive data is sent to third parties without Justification**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application does not transmit sensitive data — including PII, credentials, session tokens, cryptographic keys, or payroll-specific data — to third-party services, SDKs, analytics providers, or logging platforms unless it is an explicitly necessary and documented part of the approved application architecture and privacy policy. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static analysis- Network proxy tools (Burp Suite / mitmproxy)- Access to app’s privacy policy and architecture documentation |
| **Test Data** | - Sensitive data generated by the application (e.g. login tokens, user identifiers, salary info, personal data) |
| **Test Steps** | 1. Perform **static code analysis** using MobSF/JADX:• Identify integrated third-party SDKs or services (Firebase, Crashlytics, social logins, payment processors).• Review API calls, SDK initializations, and data sent to third-party endpoints.• Flag any method sending sensitive variables to external services. 2. Conduct **dynamic network traffic analysis**:• Intercept all app network traffic via Burp Suite / mitmproxy.• Perform normal and edge-case actions (login, profile update, salary access, token generation).• Monitor HTTP/HTTPS requests to external domains (anything outside of approved backend systems).• Identify if PII, tokens, credentials, or sensitive values are transmitted to third-party services.3. Cross-check data flows against the **approved architecture diagrams and privacy policy**:• Confirm any external data sharing is explicitly listed and justified.• Validate lawful basis (under GDPR) if data transfer is personal or payroll-related. |
| **Expected Result** | - No sensitive data (PII, tokens, salary info, credentials) sent to any third-party services without explicit documentation in the privacy policy and system architecture.- Third-party SDKs only receive permitted, non-sensitive telemetry or error data.- All sensitive data transmissions are confined to the app’s own secure backend systems. |
| **Actual Result** | (To be filled post-test — e.g. JWT tokens sent to analytics.kexample.com detected during login) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if unauthorized data sharing with third parties occurs |
| **Evidence** | Example findings:• Access token included in Crashlytics event log.• User’s payroll details transmitted to unlisted third-party analytics endpoint.• Frida/mitmproxy intercept showing cleartext PII in request to external service not listed in privacy policy. |
| **Mitigation Recommendation** | - Remove or sanitize any unnecessary data sent to third-party SDKs and services.- Validate third-party SDK configurations to disable sensitive event logging.- Review privacy policy and update if third-party data sharing is necessary and justified.- Ensure lawful basis and consent obtained where required (GDPR Art. 6–7).- Regularly perform static and dynamic reviews of third-party data transmissions during pentests and security audits.- Implement data minimization: only send non-sensitive, anonymized, aggregated data to external services where required. |

**Validation that keyboard cache is disabled for Sensitive Input Fields**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that all text input fields in the mobile application processing sensitive data — including passwords, tokens, personal identifiable information (PII), salary details, or authentication factors — have keyboard caching, predictive text, and autocomplete features disabled to prevent accidental caching or unauthorized access via the keyboard cache. |
| **Pre-conditions** | - APK/IPA file available- Access to app UI forms- MobSF, JADX, apktool for static analysis (optional)- Physical or emulator device for runtime inspection |
| **Test Data** | - Input fields for:• Password• Tokens• PII (email, salary, SSN, etc.) |
| **Test Steps** | 1. Manually inspect the mobile application UI screens:• Locate text input fields handling sensitive data (login, profile update, payroll details, password reset, token entry).• Verify field behavior: no autocomplete suggestions or predictive text offered by the keyboard.2. Perform **static code review** (if possible):**Android:** Check XML layout files and code for:• android:inputType="textPassword"• android:importantForAutofill="no"• android:autofillHints="none"• android:privateImeOptions settings**iOS:** Check Storyboard/XIB or Swift code for:• textField.isSecureTextEntry = true• textField.autocorrectionType = .no• textField.autocapitalizationType = .none• textField.smartInsertDeleteType = .no (if applicable)3. Test **keyboard behavior at runtime**:• Focus each sensitive input field and check if autocomplete suggestions appear.• Input test values and verify whether the keyboard cache retains those inputs for suggestions. |
| **Expected Result** | - All sensitive text input fields disable keyboard caching, predictive text, and autocomplete.- No autocomplete suggestions shown for passwords, tokens, or sensitive PII.- Secure text input controls (SecureTextEntry, inputType="textPassword") enforced. |
| **Actual Result** | (To be filled post-test — e.g. Salary amount input field allowed autocomplete suggestions detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | Medium–High if caching enabled on sensitive fields |
| **Evidence** | Example findings:• Password input field missing android:inputType="textPassword" and allowed autocomplete suggestions• Payroll token entry text field did not set textField.isSecureTextEntry = true in iOS Swift code• Predictive text offered salary data after prior input in test scenario |
| **Mitigation Recommendation** | - Ensure all sensitive input fields disable keyboard caching and autocomplete features.- **Android:** set inputType="textPassword", autofillHints="none", importantForAutofill="no".- **iOS:** set isSecureTextEntry = true, autocorrectionType = .no, autocapitalizationType = .none.- Review and enforce secure input field practices in mobile development guidelines.- Test all sensitive fields manually during security QA and pentests.- Remove or disable caching behavior in custom keyboards or third-party input libraries if present. |

**Validation that no sensitive data exposed via User Interface**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that no sensitive data — such as passwords, PINs, tokens, cryptographic keys, or personal identifiable information (PII) — is exposed through the application’s user interface, including in input fields, system dialogs, toast messages, error messages, overlays, or debug screens. |
| **Pre-conditions** | - APK/IPA file available- Device or emulator for runtime testing- MobSF, JADX for static analysis (optional)- Frida/Objection for runtime behavior inspection |
| **Test Data** | - Test accounts, dummy sensitive data (passwords, tokens, PINs, salary info) |
| **Test Steps** | 1. Perform **manual UI walkthrough of the application**:• Inspect all login, registration, password reset, payment, and payroll-related screens.• Confirm password and PIN fields use masking (\* or dots) and are set to secure input types.2. Test for **unexpected sensitive data display**:• Trigger application errors, invalid inputs, and debugging screens.• Confirm no sensitive internal data (e.g., tokens, credentials, keys, salary data) is shown in toast messages, dialogs, or error popups.3. Perform **static code analysis** (if available):• Check for UI elements like TextView, Toast, Dialog displaying sensitive variables.• Verify that EditText input fields for passwords and PINs use android:inputType="textPassword" or isSecureTextEntry = true on iOS.4. Perform **dynamic analysis** with Frida/Objection:• Hook into UI display functions and intercept parameters passed to them.• Look for any display of sensitive data at runtime. |
| **Expected Result** | - No sensitive data (passwords, PINs, tokens, PII, salary info) visible through UI elements.- Password and PIN inputs are masked and do not display characters in clear text.- Error messages and dialogs do not leak sensitive internal values.- No debug or crash screens exposing sensitive runtime data. |
| **Actual Result** | (To be filled post-test — e.g. Access token displayed in toast message on payment error) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data is exposed via UI |
| **Evidence** | Example findings:• Login screen password input lacked android:inputType="textPassword"• Toast message displayed "Error: token 0x123abc expired" on session timeout• Salary details shown in clear text on a debug overlay for employee accounts• Frida trace revealed token string passed to Toast.makeText() |
| **Mitigation Recommendation** | - Mask all password, PIN, and sensitive input fields using android:inputType="textPassword" or isSecureTextEntry = true (iOS).- Remove or sanitize any sensitive data displayed via toast, dialogs, or debug UIs.- Disable or restrict debug overlays and verbose error messages in production builds.- Implement secure UI design review during development and regular pentests.- Include UI security controls in mobile secure coding guidelines and enforce them via code review. |

**Validation that no sensitive data is included in the backups**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that sensitive application data — such as PII, authentication credentials, cryptographic keys, session tokens, or payroll-related data — is not included in mobile operating system cloud backups (e.g., Google Drive, iCloud) by verifying that sensitive data directories, files, and preferences are excluded via platform-specific backup exclusion mechanisms. |
| **Pre-conditions** | - APK/IPA file available- Access to app manifest, file system, and runtime storage locations- MobSF, JADX for static analysis |
| **Test Data** | - Sensitive app data (tokens, credentials, salary records, PII fields) generated within the app |
| **Test Steps** | **For Android:**1. Perform **static analysis** of AndroidManifest.xml:• Verify if android:allowBackup="false" is set in the <application> tag.2. If backups are enabled, confirm that sensitive data is stored in directories excluded from backup (e.g., /data/no\_backup/ or internal cache directories).3. Inspect shared preferences and file storage locations for sensitive data.**For iOS:**1. Review app’s Info.plist file for UIApplicationExitsOnSuspend and relevant backup control keys.2. Check if sensitive files are stored in directories flagged with NSURLIsExcludedFromBackupKey = true via code or file attribute.3. Confirm sensitive data is never written to Documents or other backed-up directories.**Dynamic Testing:**4. On a test device/emulator, check runtime storage locations (/data/data/<package> on Android, app container on iOS).5. Review which directories hold sensitive data and whether those directories are part of OS backups. |
| **Expected Result** | - android:allowBackup="false" is set in AndroidManifest.xml for security-critical apps.- Sensitive files stored in Android /data/no\_backup/ or iOS locations with backup exclusion attributes.- No sensitive data stored in shared preferences, files, or directories subject to automatic OS cloud backups. |
| **Actual Result** | (To be filled post-test — e.g. User salary records stored in Android /data/data/app/shared\_prefs included in backups) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data is included in backups |
| **Evidence** | Example findings:• android:allowBackup="true" detected in AndroidManifest.xml• Salary records stored in app’s Documents folder on iOS without backup exclusion flag• No NSURLIsExcludedFromBackupKey set for sensitive files in iOS file storage code |
| **Mitigation Recommendation** | - Set android:allowBackup="false" in AndroidManifest.xml for apps handling sensitive data.- On Android, store sensitive files in /data/no\_backup/ to avoid inclusion in OS backups.- On iOS, store sensitive data in cache directories or mark files with NSURLIsExcludedFromBackupKey = true.- Avoid storing sensitive data in shared preferences or user documents directories.- Regularly review and enforce mobile secure storage and backup exclusion guidelines during code reviews and mobile app pentests.- Update secure coding policies to mandate OS backup exclusion for sensitive data categories. |

**Validation that sensitive data is cleared when Application moves to background**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application removes or obscures sensitive data (PII, passwords, tokens, salary information, session details) from visible views when the app moves to the background or is paused, preventing exposure in app switcher thumbnails, screenshots, or view caching mechanisms. |
| **Pre-conditions** | - APK/IPA file available- Test device/emulator- Access to app UI and sensitive screens (login, salary info, personal data views) |
| **Test Data** | - App-generated sensitive data (tokens, credentials, salary info, PII fields) displayed in views |
| **Test Steps** | 1. Launch the application and navigate to screens displaying sensitive data:• Login screen with entered password/token• Salary or payroll details• User profile data2. Move the application to background (multitask/app switcher view):• On Android: press home button / app switcher• On iOS: press home / swipe up or app switcher gesture3. Inspect the preview thumbnail shown in the system’s app switcher view:• Confirm whether sensitive data is visible in the thumbnail image or blurred/cleared.4. Review **static code (optional):Android:** check for onPause(), onStop(), onTrimMemory() handlers clearing or masking views.**iOS:** check applicationDidEnterBackground() implementation for clearing sensitive labels or enabling UIWindow.isHidden = true or similar obfuscation behavior.5. Perform **dynamic analysis**:• Use Frida/Objection to hook on backgrounding events and confirm view clearing at runtime. |
| **Expected Result** | - No sensitive data visible in app switcher thumbnails or OS task manager previews.- Sensitive views are cleared, replaced with a safe screen, or masked when the app moves to the background. |
| **Actual Result** | (To be filled post-test — e.g. Salary details visible in Android app switcher thumbnail detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data remains visible in background previews |
| **Evidence** | Example findings:• Salary screen content visible in Android app switcher thumbnail• Password input fields still populated in preview when app backgrounded on iOS• No implementation of onStop() clearing sensitive data in Android Activity logs |
| **Mitigation Recommendation** | - Implement app state handling to clear sensitive views when backgrounded:**Android:**• Override onPause(), onStop(), onTrimMemory() to blank or obfuscate sensitive UI elements.• Set a generic safe placeholder view before moving to background.**iOS:**• Use applicationDidEnterBackground() to hide sensitive views or mask the app’s main window.• Optionally enable UIApplication.shared.isProtectedDataAvailable to defer access to sensitive data.- Test app switcher previews during QA and pentests for data leaks.- Add secure UI backgrounding behavior to mobile secure coding standards. |

**Validation that sensitive data is no longer stored in memory and is cleared explicitly after usage**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that sensitive data — including passwords, authentication tokens, session secrets, salary information, and PII — is not held in memory longer than necessary, is not cached in static or global variables, and is explicitly cleared (zeroed/overwritten) immediately after use. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX for static code review- Frida/Objection for dynamic memory inspection- Access to test data (dummy credentials, tokens, PII values) |
| **Test Data** | - Sensitive app data (tokens, passwords, PII, salary values) entered via normal application usage |
| **Test Steps** | 1. Perform **static code analysis** with MobSF and JADX:• Search for sensitive data variables stored in static/global class members, singletons, long-lived objects, or caches (e.g. static String password, global session token variables).• Check if memory clearing operations like Arrays.fill(), zeroMemory(), or nullifying variables occur immediately after data usage (especially after authentication, payment, or encryption operations).2. Perform **dynamic memory analysis**:• Use Frida to hook sensitive functions (login, payment submission) and monitor memory after operations complete.• Attempt to read memory values post-authentication or post-payment and confirm sensitive values are not retained.3. Test application backgrounding and session timeouts:• Move app to background and resume — confirm sensitive values are no longer accessible in memory.• Simulate session timeouts and verify memory cleanup. |
| **Expected Result** | - Sensitive data variables are kept in memory only for the minimum time necessary.- Variables holding sensitive data are explicitly cleared (e.g., set to null, overwritten) immediately after use.- No sensitive values retained in static/global variables, caches, or long-lived objects.- Memory regions no longer hold sensitive data after logout, session expiry, or app backgrounding. |
| **Actual Result** | (To be filled post-test — e.g. Password and token retained in static global variable AuthUtils.currentToken after logout detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data remains in memory unnecessarily |
| **Evidence** | Example findings:• Static field static String lastUserPassword holding last entered password after logout.• Token value held in memory cache class TokenCache long after session expiry.• No memory clearing operations implemented in password handling methods.• Frida trace revealing salary info in memory buffer minutes after payment confirmation. |
| **Mitigation Recommendation** | - Avoid using static/global variables or in-memory caches to store sensitive data.- Hold sensitive data only in local, short-lived variables within secure scopes.- Immediately clear sensitive data from memory after use:• Overwrite char arrays (for passwords) using Arrays.fill() in Java.• Set sensitive object references to null after use.• On iOS, use secure memory zeroing functions (SecZeroMemory(), bzero() if permitted).- Ensure session cleanup processes explicitly clear sensitive memory areas.- Review and enforce secure memory handling practices in mobile development guidelines.- Include memory management checks in secure code reviews and mobile app pentests. |

**Validation that the app enforces minimum device access policy**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application enforces a minimum device access security policy by requiring the user to set a device passcode, biometric lock, or equivalent screen lock, before allowing access to sensitive application features or data (such as payroll info, PII, tokens, payment screens). |
| **Pre-conditions** | - APK/IPA file available- Test device/emulator- Device screen lock disabled for baseline check |
| **Test Data** | - App-generated sensitive data (salary records, login sessions, PII) |
| **Test Steps** | 1. Disable device passcode or biometric lock on test device/emulator.2. Launch the application and attempt to access sensitive features:• Salary details screen• Profile with PII• Payment or token generation functions3. Observe application behavior:• Check whether the app displays a security enforcement warning or denies access to sensitive areas if no passcode is configured.4. If available, verify **application security policy settings** (e.g. MDM enforced apps, corporate settings in app config or manifest).5. On **iOS**, check for usage of UIApplication.shared.isProtectedDataAvailable before accessing sensitive data.6. On **Android**, review secure flag settings and screen lock status using platform APIs like KeyguardManager.isDeviceSecure(). |
| **Expected Result** | - Application should enforce a device passcode or screen lock requirement before granting access to sensitive app functions.- If no screen lock is set, the app must notify the user and optionally restrict access to protected features. |
| **Actual Result** | (To be filled post-test — e.g. App allowed access to salary details on device with no passcode detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | Medium–High (depending on app sensitivity and business risk) |
| **Evidence** | Example findings:• No enforcement of device passcode for accessing payroll module.• Salary and PII accessible while device screen lock disabled.• No check for KeyguardManager.isDeviceSecure() on Android app.• iOS app reads protected data while isProtectedDataAvailable was false. |
| **Mitigation Recommendation** | - Enforce device passcode or biometric screen lock before accessing sensitive app features:**Android:** use KeyguardManager.isDeviceSecure() API to verify screen lock presence.**iOS:** check UIApplication.shared.isProtectedDataAvailable before handling sensitive data.- Prompt user to set a passcode if absent, or disable sensitive functionality until the device is secured.- Enforce corporate MDM device security profiles (passcode policies) for enterprise deployments.- Document this control in the app’s security policy and enforce it via regular code reviews and pentests. |

**Validation That No Sensitive Data Is Stored Locally on Device and Is Retrieved Securely from Remote Endpoints When Needed**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application does not persistently store sensitive data (PII, authentication credentials, session tokens, salary information, cryptographic keys) on the device’s local file system, preferences, databases, or caches, and instead retrieves data securely from authorized remote endpoints as needed, holding it transiently in memory for the duration of use only. |
| **Pre-conditions** | - APK/IPA file available- Test device/emulator with file system access- MobSF, JADX, apktool for static analysis- Frida/Objection for dynamic memory inspection- Burp Suite / mitmproxy for network interception |
| **Test Data** | - Sensitive data elements (user tokens, PII, salary details, encrypted records) generated through normal app operations |
| **Test Steps** | **Static Analysis:**1. Use MobSF/JADX to scan for local storage API usage:• SharedPreferences, SQLiteDatabase, FileOutputStream, NSUserDefaults, CoreData.2. Review code for any storage of sensitive values in local files, cache directories, or persistent storage areas.**Dynamic Analysis:**3. Install app on test device/emulator.4. Perform sensitive app actions (login, view salary, generate tokens).5. Use ADB shell (Android) or iOS file browser to browse the app’s sandbox and shared storage directories:• Android: /data/data/<package>/, /sdcard/, /storage/emulated/0/• iOS: App container directories6. Verify that no sensitive data files, logs, cache records, or preference files persist sensitive data.**Network Analysis:**7. Use Burp Suite / mitmproxy to intercept app traffic.8. Confirm sensitive data is retrieved securely (TLS 1.2/1.3 with certificate validation) on demand from remote APIs.9. Validate no sensitive data preloads or excessive caching from API calls.**Memory Analysis (optional):**10. Use Frida to hook into runtime memory and verify sensitive data held only in short-lived variables and cleared post-use. |
| **Expected Result** | - No sensitive data stored persistently on device storage (files, preferences, local DB).- Sensitive data retrieved securely from remote endpoints only when needed.- Data held in memory transiently and cleared after use.- All sensitive communication secured via TLS 1.2/1.3 with proper certificate validation. |
| **Actual Result** | (To be filled post-test — e.g. Salary details cached in SQLite database table detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data stored locally or insecurely cached |
| **Evidence** | Example findings:• JWT tokens stored in SharedPreferences• User PII written to config.db SQLite table• Salary records cached in Documents/ folder on iOS• API responses containing sensitive data pre-fetched and stored in files |
| **Mitigation Recommendation** | - Remove persistent local storage of sensitive data.- Retrieve sensitive data on-demand via secure, authenticated API requests over TLS 1.2/1.3.- Hold sensitive data only in transient memory variables and explicitly clear memory after use.- Avoid caching sensitive API responses locally.- Disable automatic OS backup for transient memory or sensitive data (Android allowBackup="false", iOS NSURLIsExcludedFromBackupKey).- Regularly review data flow designs in mobile pentests and secure code reviews.- Document this principle in your mobile app data handling policy and development guidelines. |

**Validation That Locally Stored Sensitive Data Is Encrypted Using a Hardware-Backed Key With User Authentication**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that if the mobile application stores sensitive data locally (PII, tokens, salary data, credentials), the data is encrypted at rest using keys stored in hardware-backed secure storage (Android Keystore or iOS Keychain) — and that key access is gated by device-level user authentication (passcode, biometric, device lock) before use. |
| **Pre-conditions** | - APK/IPA file available- MobSF, JADX, apktool for static analysis- Test device with secure lock screen and biometric enabled- Frida/Objection for runtime analysis |
| **Test Data** | - Locally stored sensitive data (if any): tokens, credentials, salary records |
| **Test Steps** | **Static Analysis:**1. Use MobSF/JADX to inspect data storage operations:• Check for use of encrypted file storage APIs or database encryption libraries.• Verify encryption keys are retrieved via secure system APIs:• **Android:** AndroidKeyStore with KeyGenParameterSpec.Builder.setUserAuthenticationRequired(true) and setIsStrongBoxBacked(true) if available.• **iOS:** SecItemAdd() with kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly or kSecAttrAccessibleWhenUnlockedThisDeviceOnly.**Dynamic Analysis:**2. Use Frida/Objection to monitor file encryption and decryption operations at runtime.3. Attempt unauthorized key access without device authentication (e.g. disable lock screen, test in insecure mode).4. Use ADB shell or file system tools to check for encrypted files — verify storage locations, file encryption status, and key access control behavior.5. If possible, attempt to decrypt stored data after device lock without re-authentication and verify denial of access. |
| **Expected Result** | - Sensitive data at rest is encrypted using a hardware-backed key.- Key usage requires device authentication (passcode/biometrics).- Encryption keys are non-exportable and cannot be accessed or extracted via user-space APIs.- No sensitive data accessible on device without proper user authentication.- Encrypted files or databases remain unreadable without device unlock or biometric verification. |
| **Actual Result** | (To be filled post-test — e.g. Tokens encrypted using symmetric key stored in Android Keystore without user authentication requirement detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if encryption is weak, software-only, or without enforced authentication |
| **Evidence** | Example findings:• KeyGenParameterSpec.setUserAuthenticationRequired(false) used in Android Keystore setup.• Sensitive SQLite database encrypted with a key stored in app memory instead of Keystore.• No kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly protection applied to iOS Keychain items.• Frida interception showing key access permitted without passcode or biometric challenge. |
| **Mitigation Recommendation** | - Enforce hardware-backed encryption for all locally stored sensitive data:**Android:**• Use AndroidKeyStore with KeyGenParameterSpec.setUserAuthenticationRequired(true) and, if possible, setIsStrongBoxBacked(true).**iOS:**• Store sensitive items in Keychain with kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly or kSecAttrAccessibleWhenUnlockedThisDeviceOnly.- Avoid software-only key storage or keys in app memory.- Explicitly trigger biometric/device authentication on key usage where feasible.- Review and enforce mobile secure storage policies via static code reviews, secure coding practices, and pentests.- Disable local storage of sensitive data wherever possible (per your previous test case control). |

**Validation That the Application Educates Users About Processed PII and Security Best Practices**

**[Not a test case useful for additional user security alone]**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application transparently educates the user about the categories of sensitive personal data (PII) it processes, the purposes for which it is collected, and the security best practices the user should follow while using the app to protect their own data and privacy. |
| **Pre-conditions** | - APK/IPA file available- Access to app onboarding, settings, help, and privacy policy sections |
| **Test Data** | - No special test data required — app UI content inspection |
| **Test Steps** | 1. Perform a **manual review of the application’s onboarding process and privacy notices**:• Verify if the app explicitly informs the user about what sensitive data it collects (e.g. salary, personal details, identification numbers, credentials).• Confirm if the app explains how and why this data is used, stored, or transmitted.2. Check the **privacy policy and security help sections**:• Ensure there’s a dedicated section on user security best practices (e.g. setting device passcodes, avoiding rooted/jailbroken devices, not sharing login credentials, logging out after use).• Verify if the app warns users against suspicious activity, phishing, or using public/untrusted networks for sensitive transactions.3. Confirm whether the app offers **contextual security tips** in sensitive sections (e.g. warning before viewing salary data or managing credentials).4. Verify accessibility:• The data usage notice and security advice should be easily accessible via settings, help, or onboarding — not buried deep in legal text. |
| **Expected Result** | - The app provides clear, user-friendly explanations about processed sensitive data categories and purposes.- It communicates actionable security best practices for users.- Privacy policy or help sections are accessible and understandable, without legal jargon only.- Users are warned about security risks in relevant in-app contexts (e.g. sensitive transactions). |
| **Actual Result** | (To be filled post-test — e.g. No visible privacy notice or user security guidance detected in app UI or settings) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | Medium–High depending on app sensitivity and regulatory obligations |
| **Evidence** | Example findings:• No privacy notice explaining sensitive data categories found in onboarding or settings.• No security best practice advice present in app or help section.• Privacy policy page inaccessible or hidden behind legal terms only.• No warnings shown before exposing salary/PIN details in app UI. |
| **Mitigation Recommendation** | - Add a clear, accessible privacy notice explaining:• Categories of personal/sensitive data processed.• Why and how the data is used, stored, and transmitted.- Include a user security awareness/help section with advice such as:• Set a strong device passcode.• Avoid using public/untrusted Wi-Fi.• Beware of phishing attempts.• Regularly update the app.• Never share login credentials.- Add contextual security tips in sensitive areas of the app UI.- Ensure all information is clear, concise, and in plain, user-friendly language.- Regularly review and update privacy and security notices to comply with GDPR, ISO 27701, and OWASP MASVS privacy guidance. |

**Validation That App Local Storage Is Wiped After Excessive Failed Authentication Attempts**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application wipes or securely deletes all locally stored sensitive data after a predefined number of consecutive failed authentication attempts. |
| **Pre-conditions** | - APK/IPA file available- Test device or emulator- MobSF, JADX for static analysis- Frida/Objection optional- Device storage access (ADB or iOS file browser) |
| **Test Data** | - Valid test user account- Incorrect credentials (test passwords / PINs) |
| **Test Steps** | 1. Install and configure the app with valid credentials.2. Confirm presence of sensitive local data after normal use (tokens, PII, salary info, preferences).3. Enter incorrect credentials consecutively (e.g. 5–10 times) to trigger failed login attempts.4. Observe application behavior at threshold (warning, wipe action).5. After threshold reached, inspect app storage directories:• Android: /data/data/<package>/, /sdcard/• iOS: App container folders.6. Confirm whether sensitive files, databases, tokens, cache, and preferences are securely deleted or cleared.7. Optionally, inspect static code for local wipe logic after failed attempts. |
| **Expected Result** | - Sensitive data is securely wiped after reaching the maximum allowed failed login attempts.- No local files, cached tokens, encrypted data, or preferences remain accessible.- Optionally, app shows a wipe confirmation or warning prompt before deletion. |
| **Actual Result** | (To be filled post-test — e.g. Tokens and salary data persisted in files and database after 10 failed logins) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data persists after the allowed failed login threshold |
| **Evidence** | Example findings:• Token.json persisted in /data/data/app/files/ after 10 failed logins.• Payroll.db database remained intact.• No delete operation detected in onLoginFailure() handler or similar logic.• No user warning before wipe missing. |
| **Mitigation Recommendation** | - Implement a local wipe policy:• Define max failed login attempts (e.g. 5, configurable)• On threshold, securely delete sensitive local data: files, preferences, databases, and encryption keys• Optionally notify the user before wipe• Use secure deletion APIs appropriate for Android/iOS- Document the wipe policy in app security and coding standards- Verify wipe functionality in QA, security code reviews, and pentests |

**Verifying Data encryption on the network**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that all data transmitted over the network by the application is encrypted using a secure channel (TLS 1.2/1.3) and that secure transport is enforced consistently across all network communications, including API calls, authentication requests, and third-party service interactions. |
| **Pre-conditions** | - APK/IPA file available- Test device or emulator- Burp Suite or mitmproxy configured for TLS interception- App proxy trust configured (if testing TLS pinning bypass scenarios) |
| **Test Data** | - Sensitive data transmitted by app (credentials, tokens, PII, salary data) |
| **Test Steps** | 1. Launch the app on a test device with an intercepting proxy (Burp Suite / mitmproxy) configured.2. Capture and review all network traffic while performing typical app actions (login, salary retrieval, data submission).3. Confirm that:• All API requests use HTTPS (no plaintext HTTP traffic).• TLS protocol version is 1.2 or 1.3.• Valid, trusted, and verified server certificates are used.• Certificate pinning is enforced (if applicable).4. Attempt to intercept and downgrade traffic (disable HTTPS in proxy). Confirm app blocks insecure connections.5. Review static code and manifest files (if available) for:• Hardcoded insecure endpoints (http://)• Network security configuration (Android Network Security Config, iOS App Transport Security rules).6. Test all third-party SDK calls for secure transport (analytics, crash reporting, social login). |
| **Expected Result** | - All network traffic uses HTTPS/TLS 1.2 or 1.3.- No plaintext HTTP traffic observed.- Valid server certificates presented and verified.- App enforces secure connections consistently for all internal and third-party services.- Optional: Certificate pinning implemented on critical endpoints. |
| **Actual Result** | (To be filled post-test — e.g. Unencrypted HTTP traffic detected when accessing salary report API) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if any unencrypted or poorly secured traffic is detected |
| **Evidence** | Example findings:• HTTP call to http://api.example.com/salary exposed sensitive data• TLS 1.0 connection accepted for token exchange API• Third-party analytics SDK transmitting PII over HTTP• Missing certificate validation in HTTP client configuration |
| **Mitigation Recommendation** | - Enforce HTTPS/TLS 1.2+ for all network communications in code and app config.- Implement Android Network Security Configuration / iOS App Transport Security to block HTTP.- Remove any hardcoded HTTP URLs.- Verify and trust server certificates properly.- Implement certificate pinning for high-value endpoints (authentication, payroll APIs).- Regularly pentest and audit app network traffic for cleartext data exposure.- Review third-party SDK configurations to confirm secure transport is enforced. |

**Verifying TLS Settings Against Current Best Practices**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application’s TLS implementation follows current security best practices by using only secure protocol versions (TLS 1.2 and 1.3), strong cipher suites, valid certificates, proper certificate validation, and optionally, certificate pinning on critical endpoints. |
| **Pre-conditions** | - APK/IPA file available- Test device/emulator- Burp Suite Pro or mitmproxy with TLS inspection- Access to application backend endpoints or known API URLs |
| **Test Data** | - Application-generated sensitive data (tokens, salary details, authentication requests) |
| **Test Steps** | 1. Configure Burp Suite or mitmproxy to intercept and analyze TLS traffic from the app.2. Capture and inspect TLS handshake details for every HTTPS request:• Confirm TLS protocol version (should be 1.2 or 1.3).• Inspect server certificate validity (correct domain, expiration date, chain of trust).• Review accepted cipher suites for strong, modern ciphers (AES-256-GCM, ChaCha20-Poly1305).3. Attempt to force the app to connect using deprecated protocols (TLS 1.0, 1.1) by configuring a downgrade proxy — app should reject insecure handshakes.4. Check if weak or deprecated ciphers (3DES, RC4, null ciphers) are permitted in any handshake.5. Review static code or config (if available):**Android:** networkSecurityConfig.xml, Retrofit/OkHttp TLS settings.**iOS:** App Transport Security settings in Info.plist.6. Check whether certificate pinning is in place on critical endpoints (e.g., auth, payment APIs).7. Optionally use SSL Labs or testssl.sh against backend domains for external TLS hardening review. |
| **Expected Result** | - Only TLS 1.2 and TLS 1.3 connections are accepted.- Only modern, strong cipher suites used (e.g., AES-256-GCM, ChaCha20-Poly1305).- Deprecated ciphers (3DES, RC4, null) and protocols (SSLv3, TLS 1.0/1.1) are rejected.- Certificates are valid, trusted, and properly verified by the client.- Certificate pinning enabled on sensitive endpoints (if applicable). |
| **Actual Result** | (To be filled post-test — e.g. TLS 1.0 connection accepted to salary report API; weak cipher TLS\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA negotiated) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if deprecated TLS versions or weak ciphers accepted; Medium if no pinning or minor misconfigurations |
| **Evidence** | Example findings:• TLS 1.0 connection successfully established with payroll API endpoint.• Weak cipher TLS\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA accepted.• Certificate validity checks disabled in Retrofit client.• No certificate pinning configured for authentication endpoint. |
| **Mitigation Recommendation** | - Enforce TLS 1.2 and TLS 1.3 only.- Disable TLS 1.0, 1.1, SSLv3.- Restrict to strong cipher suites (AES-256-GCM, AES-128-GCM, ChaCha20-Poly1305).- Validate all server certificates properly, with full chain validation.- Implement certificate pinning for critical endpoints (authentication, financial APIs).- Regularly test TLS configurations using tools like testssl.sh, SSL Labs, or Burp’s Scanner.- Review and enforce secure network transport configurations in app code, manifest files, and backend server settings. |

**Verifying X.509 Certificate Validation on Secure Channels**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application correctly validates the X.509 certificate presented by the remote endpoint during TLS handshake and only accepts certificates signed by a trusted CA, with proper certificate chain validation and domain matching. |
| **Pre-conditions** | - APK/IPA file available- Test device/emulator- Burp Suite Pro / mitmproxy with interception and custom CA injection- Access to backend API domains |
| **Test Data** | - Normal app usage that triggers network requests (authentication, data retrieval, payroll actions) |
| **Test Steps** | 1. Install and trust Burp Suite / mitmproxy CA on the test device (if possible).2. Intercept app traffic via proxy and present a self-signed or proxy-generated certificate for a backend API call.3. Attempt to initiate a TLS connection.4. Observe if the application rejects the connection or proceeds with the invalid certificate.5. Review static code (if possible):**Android:** Retrofit/OkHttp TLS settings, TrustManager implementation, NetworkSecurityConfig file.**iOS:** URLSession delegate methods for didReceive challenge and trust evaluations.6. Optionally test known invalid cert scenarios:• Expired certificate• Certificate with invalid CN/SAN• Certificate signed by an untrusted CA |
| **Expected Result** | - App must verify the X.509 certificate chain.- Only certificates signed by a trusted CA (OS trust store or app-defined trust anchors) should be accepted.- Self-signed, expired, or invalid certificates must be rejected.- The certificate CN/SAN should match the requested domain name.- Optional: Certificate pinning implemented for high-risk APIs. |
| **Actual Result** | (To be filled post-test — e.g. App accepted self-signed proxy certificate for salary API endpoint) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if invalid certificates are accepted or CA validation is bypassed |
| **Evidence** | Example findings:• App accepted Burp proxy-generated certificate without error.• No domain verification in URLSession delegate.• Retrofit TrustManager configured to trust all certificates (TrustAllManager) detected.• No certificate pinning on authentication endpoint. |
| **Mitigation Recommendation** | - Enforce X.509 certificate chain validation using OS trust stores.- Reject self-signed and untrusted certificates.- Confirm CN/SAN domain match.- Avoid accepting certificates signed by untrusted root or intermediate CAs.- Implement certificate pinning for critical endpoints (auth, payroll APIs).- Use NetworkSecurityConfig on Android and App Transport Security on iOS to enforce trust policies.- Test certificate validation behavior during every pentest and secure code review. |

**Verifying Certificate Pinning or Custom CA Enforcement**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the application either uses its own internal certificate authority (CA) for TLS connections or pins the endpoint’s X.509 certificate or public key, rejecting connections to endpoints that present a different certificate or key — even if signed by a valid, trusted CA. |
| **Pre-conditions** | - APK/IPA file available- Test device/emulator- Burp Suite Pro or mitmproxy with TLS interception and custom cert injection- Access to backend endpoint domains |
| **Test Data** | - Sensitive API transactions (login, payroll retrieval, salary submissions, token management) |
| **Test Steps** | 1. Install and trust Burp/mitmproxy CA cert on test device (if necessary).2. Intercept and swap backend endpoint certificate (present a valid CA-signed cert for the correct domain, but with a different key or cert chain).3. Attempt TLS connection and API requests via the app.4. Observe whether the app detects the mismatch and blocks the connection.5. Review app static code (if possible):**Android:** OkHttp/Retrofit CertificatePinner implementation or NetworkSecurityConfig with pin-set definitions.**iOS:** URLSession delegate implementing certificate / public key pinning callbacks.6. Confirm pinning is applied for sensitive or critical APIs (login, payroll, token management). |
| **Expected Result** | - Application rejects TLS connections where the server certificate or public key does not match the pinned value(s), regardless of whether it’s signed by a trusted CA.- No sensitive data is transmitted to untrusted or non-matching endpoints.- App enforces pinning on all critical endpoints consistently. |
| **Actual Result** | (To be filled post-test — e.g. App accepted Burp’s valid CA-signed cert for payroll API despite key mismatch) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if pinning is missing or improperly enforced |
| **Evidence** | Example findings:• App accepted TLS handshake with Burp’s valid CA-signed cert for payroll endpoint.• No OkHttp CertificatePinner object configured in network client.• No pinning delegate implementation in iOS URLSession.• No Android NetworkSecurityConfig pin-set policy defined. |
| **Mitigation Recommendation** | - Implement certificate or public key pinning for all sensitive endpoints:**Android:** Use OkHttp’s CertificatePinner or NetworkSecurityConfig pin-set.**iOS:** Implement URLSessionDelegate certificate pinning via public key hash checks.- Reject connections to any endpoint offering a non-pinned cert, even if signed by a trusted CA.- Regularly update pinned values during certificate renewals or rotations.- Document and test pinning behavior in security QA and mobile app pentests.- For enterprise/private apps, optionally use a private CA and verify against it. |

**Verifying That Critical Operations Don’t Rely on a Single Insecure Channel**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application does not rely solely on a single insecure communication channel (such as email or SMS) for critical operations such as account enrollments, password recovery, or multi-factor authentication — and instead uses secure, multi-factor or in-app mechanisms for sensitive actions. |
| **Pre-conditions** | - APK/IPA file available- Test user accounts with recovery and enrollment options configured- Access to test device with SMS and email clients |
| **Test Data** | - Valid and invalid email addresses / mobile numbers- Test recovery and enrollment triggers |
| **Test Steps** | 1. Trigger account enrollment / password reset processes via email or SMS recovery options.2. Check whether the entire process (from initiation to confirmation) relies solely on an email link or SMS code without additional in-app or out-of-band verification (e.g. biometric auth, secondary email, device-bound push confirmation).3. Verify if app enforces in-app validation of the code or link (not via browser-only confirmation).4. Observe whether insecure links or codes sent over email/SMS can be intercepted, replayed, or brute-forced.5. Attempt known SMS interception or email compromise test cases if permitted (via proxy tools or device SMS listeners). |
| **Expected Result** | - App does not rely exclusively on email or SMS for sensitive account operations.- Multi-factor, in-app validation, or additional controls (biometric, device binding, knowledge factor) are used.- Password reset and account enrollment processes involve multiple independent secure steps.- Links or codes expire quickly and have limited validity. |
| **Actual Result** | (To be filled post-test — e.g. Password reset completed with SMS code alone, no in-app verification) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if critical operations rely on single insecure channels |
| **Evidence** | Example findings:• Password reset completed solely via an SMS OTP, no in-app or second factor required.• Enrollment link sent to email opened via browser, no device confirmation.• SMS recovery code valid for 30 minutes without usage restrictions.• No push notification or biometric check for recovery process. |
| **Mitigation Recommendation** | - Eliminate reliance on a single insecure channel for critical account operations.- Require multi-factor verification for sensitive actions (e.g. in-app PIN/biometric + SMS/email confirmation).- Enforce short expiration times and single-use restrictions for recovery codes.- Use in-app push notifications or device-bound confirmations for account recovery and enrollment approvals.- Document and enforce strong multi-channel recovery procedures in mobile app security requirements and coding guidelines.- Regularly test account recovery workflows during security QA and pentests. |

**Verifying Use of Up-to-Date Connectivity and Security Libraries**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that the mobile application uses up-to-date, actively maintained versions of all security-sensitive libraries — including networking (TLS/HTTP clients), cryptography, authentication, and secure storage libraries — and does not rely on deprecated, vulnerable, or unmaintained libraries. |
| **Pre-conditions** | - APK/IPA file available- Access to app dependencies via static code or manifest inspection (Android build.gradle, iOS Podfile/Cartfile)- Security advisory database (e.g. CVE database, Snyk, OSS Index) |
| **Test Data** | - App build configuration and dependency manifest- Public vulnerability data sources |
| **Test Steps** | 1. Decompile the app using MobSF or JADX for Android / class-dump / Hopper for iOS.2. Extract the list of bundled libraries, SDKs, and frameworks.3. Identify security-sensitive libraries:• Networking libraries (OkHttp, Retrofit, Alamofire)• TLS/SSL libraries (BoringSSL, OpenSSL, TrustKit)• Cryptography libraries (Conscrypt, libsodium)• Secure storage frameworks (EncryptedSharedPreferences, Keychain wrappers)4. Check the version numbers of each library used.5. Cross-reference against known vulnerabilities via CVE databases or tools like Snyk, OSS Index, or GitHub Dependabot security advisories.6. Review build config files:• Android: build.gradle dependencies• iOS: Podfile, Cartfile, or embedded frameworks versions7. Confirm all security libraries are actively maintained (last update within 6-12 months, no unresolved known CVEs). |
| **Expected Result** | - All security-sensitive libraries are up-to-date and maintained.- No known vulnerabilities (CVEs) affecting bundled library versions.- Deprecated, insecure, or abandoned libraries are removed or replaced.- Library updates and patch management included in app maintenance process. |
| **Actual Result** | (To be filled post-test — e.g. App uses Retrofit v2.4.0 with known CVE-2020-12345 affecting TLS handshake validation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if critical libraries are outdated or vulnerable |
| **Evidence** | Example findings:• OkHttp v3.8.0 (outdated; known vulnerabilities in pre-3.12)• TrustKit last updated 2019• CVE-2020-25643 affecting bundled Conscrypt library detected via Snyk scan |
| **Mitigation Recommendation** | - Upgrade all security-sensitive libraries to their latest stable, supported versions.- Regularly monitor CVE advisories and security bulletins for third-party libraries in use.- Include dependency updates as part of regular release and security maintenance cycles.- Replace any unmaintained or deprecated libraries with actively supported alternatives.- Document library version policies in mobile app security development standards.- Conduct periodic dependency audits as part of mobile app pentests and secure code reviews. |

**Code Quality and Build Settings**

**Verifying Application Signing and Private Key Protection**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application is signed with a valid, non-expired, trusted signing certificate — and that the corresponding private signing key is securely protected, preferably stored in a hardware-backed keystore (e.g. HSM or Apple Developer portal protections), and not exposed or embedded in the app or build process. |
| **Pre-conditions** | - APK/IPA file available- Access to code signing details (signature metadata)- Android: access to APK signature block or apksigner tool- iOS: access to embedded provisioning profile and signature |
| **Test Data** | - The compiled mobile app package (.apk / .ipa) |
| **Test Steps** | **For Android:**1. Use apksigner verify --verbose <app.apk> to check signature scheme (v1/v2/v3) and signer certificate details.2. Confirm certificate validity (not expired, issued by correct authority).3. Inspect APK for embedded signing keys (should not exist).**For iOS:**1. Use codesign -dvv <app.ipa> or inspect embedded mobile provisioning profile.2. Confirm valid code signing identity and non-expired provisioning profile.3. Verify that no development or ad-hoc provisioning profile is used for production apps.**General Check:**4. Review build configuration (if accessible) to confirm private signing key is stored securely in a protected keystore (HSM, secure build environment, Apple Developer portal) — not in plaintext, build scripts, or local files. |
| **Expected Result** | - Application is signed with a valid, trusted, non-expired certificate.- Private signing key is securely managed and never embedded in the application package or exposed in the build process.- No self-signed or development certificates in production releases.- App signature verification passes on intended devices and stores. |
| **Actual Result** | (To be filled post-test — e.g. APK signed with expired debug keystore; no production certificate detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if invalid, debug, expired, or missing production signing detected |
| **Evidence** | Example findings:• apksigner output shows app signed with debug.keystore.• iOS provisioning profile uses ad-hoc profile for public app.• No hardware-backed key management process documented for private signing keys.• Private signing key found in plaintext in CI/CD script. |
| **Mitigation Recommendation** | - Sign production builds only with valid, non-expired certificates issued by the appropriate app store or enterprise CA.- Secure private signing keys in hardware-backed security modules (HSM, secure CI/CD vaults, or Apple Developer Portal protections).- Do not embed private keys in the app or build artifacts.- Regularly audit app signing configurations and private key management processes.- Use signature schemes v2/v3 (Android) to enforce strong integrity protection.- Ensure provisioning profiles for production iOS apps are marked as Distribution, not Development or Ad Hoc. |

**Verifying Release Build Configuration and Non-Debuggable Setting**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application is built in release mode with production-ready settings: optimizations enabled, no debug configuration present, logs and debug code removed, and the debuggable flag set to false, preventing runtime debugging on production builds. |
| **Pre-conditions** | - APK/IPA file available- Access to app manifest (Android) or build config (iOS)- ADB tools or Frida for runtime inspection |
| **Test Data** | - The compiled app binary (APK/IPA) |
| **Test Steps** | **For Android:**1. Use aapt dump badging <app.apk> or adb shell dumpsys package <package\_name> to check android:debuggable flag.2. Verify android:debuggable="false" in the APK manifest.3. Check for presence of debug logs or verbose log output using adb logcat.4. Decompile APK with MobSF/JADX to verify no debug flags, test code, or development-only configurations remain.5. Confirm ProGuard/R8 minification and code obfuscation enabled (check for minified class/method names).**For iOS:**1. Use codesign -d --entitlements :- <app> to check for get-task-allow entitlement.2. Ensure get-task-allow is set to **false** in production builds (disables debugging).3. Inspect Info.plist for any development-specific keys or configurations.4. Confirm no debug logs or dev test flags remain. |
| **Expected Result** | - Application built in release mode with android:debuggable="false" / get-task-allow=false.- Debug logs and test code removed.- ProGuard/R8 (Android) or bitcode/hardening flags (iOS) enabled where appropriate.- App rejects debugging attempts at runtime (e.g. via Frida, LLDB, or Android Studio). |
| **Actual Result** | (To be filled post-test — e.g. APK found with android:debuggable="true" and debug log statements in production code) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if app is debug-enabled or built in development mode |
| **Evidence** | Example findings:• APK manifest includes android:debuggable="true"• iOS app has get-task-allow=true in entitlements• Debug log statements (Log.d(), NSLog()) detected in production code• No ProGuard/R8 obfuscation detected (cleartext method and class names) |
| **Mitigation Recommendation** | - Build production versions of the app in release mode only.- Set android:debuggable="false" in manifest or via build.gradle config.- Set get-task-allow=false in iOS entitlements.- Remove all debug log statements and test code from production builds.- Enable ProGuard/R8 or iOS bitcode/hardening configurations to obfuscate and optimize code.- Regularly verify build artifacts during CI/CD pipeline and perform release integrity checks in pentests. |

**Verifying That Debugging Symbols Are Removed From Native Binaries**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that all debugging symbols (including symbol tables, function names, variable names, and debug strings) have been removed from native libraries and binaries (.so files for Android and Mach-O binaries for iOS) in the production build. |
| **Pre-conditions** | - APK/IPA file available- Decompilation tools: MobSF, JADX, Ghidra, Hopper, otool, readelf- Access to app’s native libraries |
| **Test Data** | - Native libraries (Android: .so files in /lib/; iOS: Mach-O binaries in app bundle) |
| **Test Steps** | **For Android:**1. Extract the APK using APKTool or unzip utility.2. Locate native libraries in /lib/ directory.3. Use readelf -S <libnative.so> and check for .symtab, .strtab, and .debug\_\* sections.4. Confirm these sections are absent or stripped.5. Optionally use Ghidra/Hopper to inspect symbol names in the binary.**For iOS:**1. Extract the IPA or use class-dump or otool -l <AppBinary>.2. Check for LC\_SYMTAB, LC\_DYSYMTAB, or debug symbol sections in Mach-O header.3. Confirm symbol names in binary have been stripped or replaced by generic/obfuscated identifiers.4. Attempt to decompile and verify whether function names and variable names are obfuscated or absent. |
| **Expected Result** | - No debugging symbols, developer comments, or function/variable names present in native binaries.- No .symtab, .strtab, or .debug\_\* sections present in production builds.- Symbolication files (ProGuard mapping, dSYM) retained separately for internal debugging if required. |
| **Actual Result** | (To be filled post-test — e.g. Android .so libraries contain unstripped symbol table and debug strings) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | Medium–High depending on sensitive logic exposed through symbols |
| **Evidence** | Example findings:• readelf -S shows .symtab and .debug\_line sections in libnative.so.• iOS binary contains cleartext method names and class identifiers in Mach-O symbol table.• Ghidra reverse engineering reveals intact function names.• Debug strings embedded in native code. |
| **Mitigation Recommendation** | - Strip debugging symbols from all native libraries before production release:**Android:** Use strip --strip-unneeded on native libraries during build process.**iOS:** Enable Strip Debug Symbols During Copy and Strip Linked Product in Xcode build settings.- Keep ProGuard mapping files (Android) and dSYM files (iOS) securely for crash reporting and debugging internally.- Validate symbol stripping as part of CI/CD release pipeline and in pentests.- Review build scripts and release signing processes to enforce this control. |

**Verifying Removal of Debugging Code, Test Routines, Backdoors, and Verbose Logging**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application production build has no remaining debugging code, test routines, backdoor functions, hidden settings menus, verbose error messages, or developer assistance code — and does not log sensitive data or verbose debugging messages in logcat (Android) or console logs (iOS). |
| **Pre-conditions** | - APK/IPA file available- Access to app manifest/config files- ADB or Frida for runtime log monitoring- Static analysis tools (MobSF, JADX, class-dump, Ghidra) |
| **Test Data** | - Sensitive actions (authentication, payroll viewing, payments, failed inputs) |
| **Test Steps** | 1. Perform static code analysis (MobSF/JADX or class-dump) to search for:• Debug flags (e.g. BuildConfig.DEBUG, if (debug) { ... })• Test routines or mock services• Hidden dev menus, backdoor keys, or special debug intents• Log.v(), Log.d(), System.out.println(), NSLog() statements2. Review manifest and config files for:• Test-only permissions• Debugging service registrations3. Use adb logcat (Android) or Xcode console (iOS) during normal and erroneous actions:• Attempt invalid logins, trigger errors, and observe if sensitive data or verbose debug output appears.4. Check for backdoor triggers or undocumented activities:• Special URLs, secret intents, custom debug gestures, or button sequences opening dev panels. |
| **Expected Result** | - No debugging, test, or developer assistance code present in production app.- No verbose log statements or sensitive data logged.- No hidden dev menus or undocumented features accessible.- Only sanitized, generic error messages shown to user and logs. |
| **Actual Result** | (To be filled post-test — e.g. Backdoor intent action com.app.DEBUGMODE found; verbose log showing tokens in logcat) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if debugging code, test hooks, backdoors, or sensitive verbose logs are present |
| **Evidence** | Example findings:• BuildConfig.DEBUG check bypass left in production• Log.d("token", tokenValue) found in AuthManager.java• Secret dev URL myapp://debugpanel accessible• Salary data shown in adb logcat during payment error |
| **Mitigation Recommendation** | - Remove all debugging and test code before production release.- Remove any hardcoded backdoor triggers, test keys, and hidden menus.- Disable verbose logging (Log.v(), Log.d(), System.out.println(), NSLog()) in production.- Enforce code minification and ProGuard/R8 obfuscation for release builds.- Ensure only sanitized, non-sensitive operational logs remain.- Include log and debug artifact removal in build and pentesting checklist.- Test final release builds to confirm no developer routines remain active. |

**Verifying Third-Party Component Security, Vulnerabilities, and Weaknesses**

| **Field** | **Detailed Content** |
| --- | --- |
| **Test Objective** | Ensure that all third-party components (libraries, SDKs, frameworks) integrated into the mobile application are identified, verified for known vulnerabilities (via public CVE databases and security advisory feeds), checked for active maintenance status, and reviewed for insecure default behaviors or weak configurations. |
| **Pre-conditions** | - APK/IPA file available- Access to app’s dependency declarations (Android build.gradle, dependencies.gradle, iOS Podfile.lock, Cartfile.resolved, embedded frameworks)- CVE databases access (NVD, CVE Details, OSS Index, GitHub Security Advisories, Snyk API) |
| **Test Data** | - Complete list of third-party libraries, SDKs, frameworks, including version numbers |
| **Test Steps** | **1️⃣ Static Analysis:**• Decompile the app using MobSF, JADX (Android) or Hopper/class-dump (iOS).• Identify all third-party libraries and frameworks included in the app.• Extract versions from manifest files or embedded libraries.• Review Android build.gradle / gradle.properties / dependencies.gradle for declared dependencies and versions.• Review iOS Podfile.lock / Cartfile.resolved / Xcode project embedded frameworks.**2️⃣ CVE Vulnerability Check:**• Cross-reference library and SDK versions against official CVE databases (CVE Details, NVD, OSS Index).• Check for high/critical CVEs affecting detected versions.• Use automated tools like Snyk CLI/API or OWASP Dependency-Check.• Review vendor-specific advisories (Firebase, OkHttp, Facebook SDK, etc).**3️⃣ Maintenance & Update Status Check:**• Review each component’s public repo (GitHub/GitLab/Bitbucket) for last commit/update date.• Flag libraries not updated in >12 months or marked deprecated.• Check open issue tracker for unresolved security reports.**4️⃣ Security Misconfiguration and Privacy Weakness Check:**• Verify default SDK behaviors (e.g., Firebase, Crashlytics logging sensitive data by default).• Inspect embedded config files or runtime behavior for insecure defaults (logging, excessive telemetry, weak TLS, cleartext logs).• Test app runtime with adb logcat (Android) or Xcode console (iOS) for third-party logs.**5️⃣ License & Compliance Check (Optional but Recommended):**• Confirm third-party licenses (Apache-2.0, MIT, GPL) to ensure no compliance violations. |
| **Expected Result** | - All third-party libraries and SDKs accurately identified.- No known unpatched critical/high CVEs affecting any component version.- No deprecated, abandoned, or unsupported libraries.- No insecure SDK defaults or privacy-invading behaviors (e.g., logging PII).- All third-party SDKs securely configured.- Optionally: all third-party licenses validated and compliant. |
| **Actual Result** | (To be filled post-test — e.g. Retrofit v2.3.0 detected with CVE-2019-12202; analytics SDK logging PII detected during login failure) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if critical CVEs or unmaintained/insecure libraries detected; **Medium** for minor weaknesses or misconfigurations |
| **Evidence** | Example findings:• OkHttp v3.8.0 detected with CVE-2019-9442.• Firebase SDK logging email addresses in crash reports.• Facebook SDK leaking app metadata without explicit consent.• Crashlytics logging PII and tokens.• Outdated library xyz.lib last updated in 2019, flagged for end-of-life. |
| **Mitigation Recommendation** | - Immediately patch or replace any vulnerable third-party libraries.- Replace abandoned or deprecated libraries with supported alternatives.- Disable insecure SDK defaults (e.g., logging, telemetry, open permissions).- Implement a dependency management policy with version control, CVE scanning, and approval workflows.- Automate third-party CVE and advisory monitoring (Snyk, OSS Index, DependencyTrack).- Validate third-party library security as part of every release pentest and secure code review.- Maintain a live dependency inventory and threat exposure register. |

**Verifying Exception Handling and Secure Error Management**

| **Field** | **Detailed Content** |
| --- | --- |
| **Test Objective** | Confirm that the application securely and consistently catches, handles, and logs all possible exceptions, preventing application crashes, stack trace disclosures, verbose technical messages, or sensitive data leaks to users and logs — while providing clear, generic user feedback. |
| **Pre-conditions** | - APK/IPA production build available- Test device/emulator- Debugging tools: **ADB logcat** (Android), **Xcode console** (iOS)- Network manipulation tools: **Burp Suite, mitmproxy**- Static code review tools: **MobSF, JADX, Hopper, class-dump** |
| **Test Data** | - Valid and invalid user inputs (empty fields, malformed data, large payloads)- Simulated network failures (disabled Wi-Fi, forced HTTP 500, timeouts)- Invalid API responses (tampered via proxy)- Simulated file I/O failures (e.g., no storage permission)- Simulated out-of-memory conditions (if feasible) |
| **Test Steps** | **1️⃣ Input Validation & API Exception Handling:**• Enter empty, invalid, or oversized values into form fields.• Submit malformed JSON/XML data (if possible).• Observe if exceptions are handled gracefully.• Tamper API responses (401, 500 errors, malformed JSON) via proxy — check if app gracefully handles unexpected status codes and malformed data.**2️⃣ Network Disruption Testing:**• Disable device internet (Wi-Fi and data).• Attempt login, data fetch, or payroll transaction.• Confirm app catches network exceptions (SocketTimeout, IOException, NSURLError) and displays safe, generic error messages (not stack traces).• Check logs for absence of sensitive error output.**3️⃣ File I/O Error Simulation:**• Deny storage permissions (Android) or simulate disk full error (if possible).• Trigger app functionality that writes to local storage.• Observe if exceptions are handled and appropriate messages are shown.**4️⃣ Memory & Resource Exception Handling:**• Simulate resource exhaustion (if possible) with large data uploads or repeated transactions.• Confirm OutOfMemoryError / NSException handled safely (without crash or memory dump exposure).**5️⃣ Runtime Log Monitoring:**• Use adb logcat or Xcode console during the above actions.• Check for unhandled exceptions, stack traces, sensitive error logs (tokens, salary data, internal URLs, SQL statements).• Ensure logs remain minimal and non-sensitive.**6️⃣ Static Code Analysis (Optional, if accessible):**• Use MobSF or JADX (Android) to inspect for try-catch blocks around:✔ API calls✔ JSON/XML parsing✔ File operations✔ Payment/Token processing• Confirm no Log.e(), System.out.println(), or NSLog() for exceptions in production.• Review for application-wide crash handlers:✔ Android: Thread.setDefaultUncaughtExceptionHandler()✔ iOS: NSSetUncaughtExceptionHandler() |
| **Expected Result** | - App gracefully handles all exceptions — no app crashes or stack traces displayed to users.- User receives sanitized, generic error messages (e.g. "Network unavailable. Please try again.")- No sensitive internal details exposed in errors or logs.- Log output sanitized: no tokens, credentials, PII, or internal code references.- Static code confirms try-catch presence on critical operations.- Application-wide crash handler (if feasible) redirects uncaught exceptions safely. |
| **Actual Result** | (To be filled post-test — e.g. NullPointerException triggered on payment failure without exception handler. Stack trace with salary endpoint URL shown in logcat) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if any crash, unhandled exception, or sensitive error disclosure occurs; **Medium** for non-crashing verbose logs |
| **Evidence** | Example findings:• App crashed on null API response with NullPointerException.• Verbose log showing "token=xyz123" during login error.• Stack trace printed in console showing internal file paths on API timeout.• No catch block around file write operation in payroll module.• User received "java.lang.Exception: null" error message instead of friendly text. |
| **Mitigation Recommendation** | - Implement robust try-catch-finally blocks on all critical operations.- Add global crash handler (UncaughtExceptionHandler / NSSetUncaughtExceptionHandler).- Replace technical errors with generic, user-friendly messages.- Disable all verbose or debug logs in production.- Review code for secure exception management during code reviews.- Include error handling validation in pentests and QA cycles.- Avoid logging PII, credentials, or tokens in errors or logs.- Regularly test all error scenarios, including network, I/O, and malformed data cases. |

**Verifying That Error Handling in Security Controls Denies Access by Default**

| **Field** | **Detailed Content** |
| --- | --- |
| **Test Objective** | Confirm that the application’s error handling logic within all security-critical functions (authentication, authorization, token validation, encryption, TLS verification) denies access or operations by default on error — never allowing unintended access, continuation, or insecure fallback behavior due to a handled or unhandled exception. |
| **Pre-conditions** | - APK/IPA build available- Device/emulator- Test user accounts with various permission levels- Burp Suite / mitmproxy for network tampering- Frida/Objection for runtime behavior tests (optional) |
| **Test Data** | - Valid/invalid tokens and credentials- Tampered API responses (missing parameters, invalid signatures)- Malformed requests- Network disruption scenarios |
| **Test Steps** | **1️⃣ Authentication Control Error Handling:**• Attempt to log in with missing or malformed credentials.• Send tampered or invalid tokens to APIs via proxy.• Confirm app denies access by default on errors like token parsing failures, expired tokens, or signature errors.**2️⃣ Authorization Checks:**• Access restricted resources with insufficient permissions.• Modify role or permission indicators in requests.• Check if authorization errors lead to default deny, not fallback or partial access.**3️⃣ TLS / Certificate Validation Errors:**• Use Burp/mitmproxy to present invalid or untrusted server certificates.• Confirm TLS errors deny connection — no insecure fallback to HTTP or acceptance of invalid certs.**4️⃣ Input Validation & Encryption Error Cases:**• Send malformed encrypted payloads to APIs.• Check that decryption or parsing errors trigger default access denial, not fallback behavior.**5️⃣ Exception Simulation:**• Disable network mid-request (simulate API failure).• Inject malformed data via proxy.• Confirm app handles errors safely — no unintended operation or success defaults. |
| **Expected Result** | - All errors in security control logic deny access by default.- No fallback to insecure behavior or partial access.- User receives clear, generic "Access denied" or equivalent error messages.- App logs sanitized errors (if necessary) without exposing sensitive stack traces or details.- No operation proceeds if any validation, token check, encryption, or authorization control fails. |
| **Actual Result** | (To be filled post-test — e.g. Authorization check failed silently and allowed salary API access without role validation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if any failure in security logic results in unintended access or operation continuation |
| **Evidence** | Example findings:• Token parsing error bypass allowed access to payroll API.• Invalid certificate accepted silently with fallback to HTTP.• Role validation error allowed unauthorized feature access.• Missing field in request caused 500 error but operation completed on server. |
| **Mitigation Recommendation** | - Design security controls to **deny by default** on any error or unexpected condition.- Log errors securely and return clear, sanitized messages to users.- Disable fallback behaviors for TLS, token parsing, or role checks.- Enforce strict input validation with rejection on failure.- Test and simulate all error scenarios in security QA and pentests.- Implement fail-safe defaults in all security-critical logic.- Include this requirement in secure coding and mobile architecture guidelines.- Validate control resilience via automated and manual test cases. |

**Verifying Secure Memory Handling in Java/Kotlin (Android)**

| **Field** | **Detailed Content** |
| --- | --- |
| **Test Objective** | Verify that the application securely manages memory in Java/Kotlin by clearing sensitive data after use, avoiding memory leaks, preventing unnecessary data persistence, and never exposing sensitive information through long-lived variables or logs. |
| **Pre-conditions** | - APK build available- MobSF, JADX for static analysis- ADB logcat and Frida/Objection for runtime inspection |
| **Test Data** | - Sensitive data handled by the app (passwords, tokens, salary info, cryptographic keys) |
| **Test Steps** | **1️⃣ Static Code Analysis (if accessible via MobSF/JADX):**• Search for use of static or global variables holding sensitive data.• Identify use of String for sensitive data (which cannot be zeroed after use).• Check for proper nullification of sensitive objects after use (e.g., Arrays.fill(charArray, '\0')).• Ensure no long-lived objects (e.g. Singletons, global caches) hold sensitive data unnecessarily.**2️⃣ Runtime Analysis:**• Perform normal operations (login, payment, salary retrieval).• Use adb logcat and Frida to monitor memory behavior during and after operations.• Confirm sensitive data not logged.• Verify garbage collection removes sensitive objects when out of scope (if possible via heap inspection).**3️⃣ Sensitive Data in Memory:**• Confirm passwords, tokens, keys are stored in char[] or securely handled structures (not String).• Check explicit clearing of char arrays after authentication and encryption.• Review token and key variables nullified or zeroed after use.**4️⃣ Exception Scenarios:**• Simulate API errors, transaction aborts, and ensure sensitive in-memory objects are cleared and not held beyond intended scope. |
| **Expected Result** | - No sensitive data stored in static/global variables.- No sensitive data kept in memory longer than necessary.- Use of char[] for passwords.- Sensitive char arrays or byte arrays explicitly cleared (e.g., Arrays.fill(charArray, '\0')).- No sensitive data written to logs.- No memory leaks from long-lived object references containing PII or tokens.- App denies access or clears sensitive memory on errors. |
| **Actual Result** | (To be filled post-test — e.g. Token held in static variable TokenManager.currentToken after logout) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if sensitive data persists in memory unnecessarily or is logged; **Medium** for minor memory management issues |
| **Evidence** | Example findings:• String password = "user123" used directly without clearance.• Static currentUserToken remained after logout.• Logcat entry with token value on failed login.• No clearing of char[] password arrays after authentication process.• Use of singletons for PII storage. |
| **Mitigation Recommendation** | - Never store sensitive data in static or global variables.- Use char[] for passwords/tokens and clear immediately after use with Arrays.fill().- Avoid long-lived caches or singletons for sensitive objects.- Remove all sensitive log statements.- Implement memory-clearing logic in finally blocks where possible.- Review object lifetimes and potential leaks during secure code reviews and pentests.- Include memory safety checks in QA test cases for sensitive features. |

**Verifying Toolchain Security Hardening Features (Expanded Test Steps)**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application uses available toolchain security hardening features — including ProGuard/R8 minification, stack protection, PIE support, ARC (iOS), symbol stripping, ASLR/DEP/CFI for native code — to prevent reverse engineering and runtime memory exploitation. |
| **Pre-conditions** | - Final production APK or IPA build available- Device/emulator access- Installed tools: MobSF, JADX, readelf, otool, nm, Frida (optional) |
| **Test Data** | - APK/IPA binary- Native .so files (if present)- Mach-O binaries (for iOS apps) |
| **Test Steps** | **Android:**1️⃣ Decompile the APK using **MobSF** or **JADX**.• Inspect decompiled class, method, and variable names.• Confirm names appear obfuscated (e.g., a.b(), x.y.z()) and not readable (e.g., TokenManager.getToken()).2️⃣ Extract .so native libraries from /lib/ directory within the APK.• Run readelf -sW libnative.so | grep stack\_chk\_fail to check for stack protection symbols.• If absent — stack canary protections not compiled in.3️⃣ Run readelf -h libnative.so.• Check the Type: field under ELF header.• Must be DYN (Shared object) to confirm PIE (position-independent executable) support.4️⃣ Upload the APK to **MobSF** and run a static scan.• Review report sections for:✔ ASLR enabled✔ DEP (NX) enabled✔ PIE support present for native binaries**iOS:**1️⃣ Use otool -oV AppBinary | grep objc\_release.• Minimal explicit retain/release calls implies Automatic Reference Counting (ARC) is enabled.• A large number of manual calls indicates ARC not fully adopted.2️⃣ Run otool -hv AppBinary.• Locate the PIE flag in Mach-O header.• Must read PIE = YES for position-independent code.3️⃣ Run otool -Iv AppBinary | grep stack\_chk.• Confirm both \_\_stack\_chk\_guard and \_\_stack\_chk\_fail symbols are present — indicating stack canary protections active.4️⃣ Run nm -gU AppBinary.• Look for human-readable function or class names.• In a secure release build, no debug symbols or internal class names should be visible.5️⃣ (Optional) Run `otool -l AppBinary |
| **Expected Result** | - Android: ProGuard/R8 minification active, stack protection symbols present in native libraries, PIE enabled, ASLR/DEP enforced via MobSF.- iOS: ARC active, PIE flag set, stack canary symbols present, debug symbols stripped. |
| **Actual Result** | (To be filled after testing — e.g. ProGuard not applied, native library lacks stack protection symbols) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if any major hardening control missing in release build |
| **Evidence** | Example findings:• Decompiled APK exposes UserManager.getToken() in clear.• readelf shows no \_stack\_chk\_fail symbol in native .so• iOS otool reveals PIE flag disabled.• nm reveals developer class symbols like TokenHandler in release binary. |
| **Mitigation Recommendation** | - Android:• Enable ProGuard/R8 for minification and obfuscation.• Compile native code with -fstack-protector-strong flag.• Ensure PIE support by confirming Type: DYN.• Validate ASLR/DEP in MobSF reports.- iOS:• Enforce ARC for all Objective-C modules.• Ensure Enable PIE set to YES in Xcode.• Compile with -fstack-protector-strong for native components.• Strip all debug symbols for release.• Confirm via otool and nm before release publishing. |

**Platform Interaction**

**Verifying Minimal and Necessary Permission Requests**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application requests only the minimal set of platform permissions necessary for its legitimate business functionality, and no excessive, unused, or privacy-sensitive permissions are requested without justified need. |
| **Pre-conditions** | - APK/IPA production build available- Device/emulator access- MobSF, JADX, ADB tools (Android)- Hopper, class-dump (iOS) |
| **Test Data** | - The final mobile application binary (APK/IPA)- The app’s documented feature scope or privacy policy (if available) |
| **Test Steps** | **Android:**1️⃣ Decompile APK with **MobSF** or **JADX**.• Review AndroidManifest.xml permissions.• List all <uses-permission> entries.2️⃣ Cross-check each permission against the app’s actual documented features and expected functionality.3️⃣ Identify unnecessary, excessive, or sensitive permissions (e.g. READ\_SMS, ACCESS\_FINE\_LOCATION, RECORD\_AUDIO) if unrelated to core features.4️⃣ Install the app on a test device and monitor permission requests via **Settings > App Info > Permissions**.5️⃣ Check runtime permission prompts during app use. Confirm only essential prompts triggered.**iOS:**1️⃣ Extract the IPA and open its **Info.plist**.2️⃣ Review NSLocationWhenInUseUsageDescription, NSCameraUsageDescription, etc.3️⃣ Cross-reference declared permissions with actual features provided by the app.4️⃣ Install on test device and confirm runtime prompts align with minimal necessary access.5️⃣ Review MobSF scan report (for both Android/iOS) for overprivileged access indicators. |
| **Expected Result** | - The app declares and requests only the permissions strictly necessary for core, documented features.- No redundant or sensitive permissions (location, SMS, contacts, microphone, camera) requested unless functionally justified.- All permission requests transparently presented to users via system prompts (if runtime permissions required). |
| **Actual Result** | (To be filled after testing — e.g. App requests ACCESS\_FINE\_LOCATION but no location feature exists) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if sensitive/unjustified permissions found in production build |
| **Evidence** | Example findings:• AndroidManifest.xml requests READ\_SMS, ACCESS\_FINE\_LOCATION, RECORD\_AUDIO though the app does not use messaging, GPS, or microphone.• iOS Info.plist includes NSMicrophoneUsageDescription with no in-app recording feature.• MobSF flagged READ\_CONTACTS permission as high-risk, no feature linked to contacts access. |
| **Mitigation Recommendation** | - Remove all permissions not explicitly required for core application functionality.- Review all <uses-permission> and Info.plist keys against a minimal access principle.- Avoid requesting sensitive permissions (location, SMS, camera, microphone, contacts) unless justified by documented features and privacy policy.- Re-test permissions on every release.- Include permission audits in regular pentesting and secure code reviews.- Inform users of permissions and their purpose during onboarding and via privacy policy (for GDPR compliance). |

**Verifying Input Validation and Sanitization for All External Sources**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that all inputs from external sources (user interface, IPC mechanisms like Intents, custom URLs, broadcast receivers, and network data) are properly validated for type, length, range, format, and context — and if necessary, sanitized before being processed by the application. |
| **Pre-conditions** | - APK/IPA build available- Device/emulator access- MobSF, JADX (Android)- Frida, Objection (for runtime testing)- Burp Suite / mitmproxy for network input tampering |
| **Test Data** | - Malformed, oversized, invalid, special-character, and unexpected inputs- Tampered Intents and custom URLs- Modified network API responses |
| **Test Steps** | 1️⃣ Review decompiled code using **MobSF** / **JADX** (Android) or **class-dump/Hopper** (iOS).• Check for validation routines around user inputs (e.g. TextUtils.isEmpty(), regex, numeric range checks).• Confirm whether received Intents, Bundles, and broadcast messages are validated before processing.2️⃣ Use **adb shell am start** to send crafted Intents (with missing, malformed, or malicious extras) to exported activities and services.• Check app behavior — must reject unsafe data.3️⃣ Test **custom URL schemes** (deep links):• Send malformed, oversized, or script-injected URLs via adb or other apps to trigger app handlers.• Confirm app validates/sanitizes input.4️⃣ Intercept and modify API responses using **Burp Suite** / **mitmproxy**:• Remove or alter expected fields, inject large strings, HTML/JavaScript payloads into JSON/XML responses.• Confirm app handles these safely without crashes or unsafe rendering.5️⃣ In UI forms, enter invalid data:• Empty values, excessively long strings, special characters, invalid formats.• Confirm proper error messages and rejection without crashes.6️⃣ Use **Frida/Objection** to hook into runtime methods (optional):• Check how untrusted data is handled and whether validation routines are invoked before processing. |
| **Expected Result** | - All external inputs are validated for presence, type, length, value range, allowed characters, and business context.- No unsafe data processed without validation.- All malformed Intents and custom URLs are rejected or safely handled.- No application crashes, unexpected behavior, or improper rendering on tampered network responses.- UI inputs enforce proper validation constraints and error messages. |
| **Actual Result** | (To be filled post-test — e.g. App accepted malformed Intent with invalid parameter and processed it without validation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if any external input processed unsafely without validation |
| **Evidence** | Example findings:• No null/length check for Intent extras in onHandleIntent().• API response with missing user\_id key crashed salary module.• Custom URL handler accepted myapp://open?<script>.• No input validation for salary input field — accepted HTML tags.• API JSON response manipulation caused unsafe app state. |
| **Mitigation Recommendation** | - Validate all external inputs for:✔ Presence (non-null, non-empty)✔ Type (integer, string, boolean, etc.)✔ Format (regex for emails, phone numbers, etc.)✔ Length and range constraints✔ Allowed characters (whitelist where possible)- Sanitize inputs before rendering to UI or storage.- Use platform-specific safe API methods (e.g. TextUtils.isEmpty(), Pattern.matches() in Android).- Validate and sanitize Intents, Bundles, and custom URLs on receipt.- Enforce robust JSON/XML schema validation on network responses.- Include validation tests in QA automation and pentests.- Add input validation coverage to secure coding guidelines. |

**Validation that no sensitive information is exposed Via IPC Mechanisms**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the mobile application does not transmit or expose sensitive data — including PII, authentication credentials, session tokens, payroll details, or cryptographic keys — through inter-process communication (IPC) mechanisms such as Intents, BroadcastReceivers, ContentProviders, or AIDL services, unless securely protected with appropriate access controls, and that any necessary IPC data exchanges are validated and documented as part of the approved application architecture. |
| **Pre-conditions** | - APK file available (Android)- MobSF, JADX, apktool for static analysis- Frida/Objection for dynamic runtime hooking- Test device/emulator with logcat and Frida access |
| **Test Data** | - App-generated sensitive data (tokens, credentials, PII, salary info) |
| **Test Steps** | 1. Perform **static code analysis** with MobSF and JADX:• Search for IPC usage: Intent, startActivity, startService, sendBroadcast, registerReceiver, ContentProvider, AIDL.• Identify what data is passed via these mechanisms.• Check for sensitive variables attached via Intent.putExtra(), Bundle, Cursor, or AIDL methods.2. Verify that sensitive data is not passed in plain Intents, Broadcasts, or unprotected ContentProviders:• Confirm no exported Activity, Service, Receiver, or ContentProvider unintentionally exposes data (exported="true" in AndroidManifest.xml without permission restrictions).3. Check if **broadcast messages** carrying sensitive data are either local-only (LocalBroadcastManager) or protected with app-specific permissions.4. Perform **dynamic analysis**:• Use Frida/Objection to hook IPC methods at runtime.• Attempt to intercept or receive broadcast Intents from outside the app’s process context.• Validate if external apps can access sensitive data via public ContentProviders or exposed IPC channels. |
| **Expected Result** | - No sensitive data (tokens, PII, salary info, keys) passed via IPC mechanisms without appropriate controls.- Sensitive IPC exchanges (if any) are explicitly secured with permissions, LocalBroadcastManager, or other isolation mechanisms.- No unintentional exposure through exported components. |
| **Actual Result** | (To be filled post-test — e.g. Payroll tokens sent via exported broadcast Intents without permission detected) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | High if sensitive data exposure via IPC occurs |
| **Evidence** | Example findings:• Intent.putExtra("token", tokenValue) sent via unprotected broadcast Intent.• ContentProvider declared as exported=true without read/write permissions.• Frida intercept showing salary data passed to an external Service via unprotected Intent. |
| **Mitigation Recommendation** | - Remove sensitive data from IPC mechanisms where possible.- Use secure alternatives:• LocalBroadcastManager for in-app broadcasts• Custom app-defined permissions for exported components- Set all exported components to exported=false in AndroidManifest.xml unless explicitly required.- If IPC is necessary, encrypt data payloads and validate calling app identities.- Conduct regular IPC misuse reviews during static code analysis and pentests.- Enforce strict IPC security controls in secure coding guidelines for mobile developers. |

**Verifying Custom URL Schemes Do Not Expose Sensitive Functionality Without Protection**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application does not expose sensitive functionality via custom URL schemes (deep links or Intent filters) unless those mechanisms are properly secured with authentication, authorization, input validation, or platform restrictions. |
| **Pre-conditions** | - APK/IPA build available- Device/emulator access- MobSF, JADX, ADB tools (Android)- Hopper, class-dump (iOS) |
| **Test Data** | - Crafted custom URLs (deep links) and Intents targeting sensitive app features |
| **Test Steps** | **Android:**1️⃣ Decompile APK using **MobSF** or **JADX**.• Review AndroidManifest.xml for <intent-filter> blocks.• Identify android:scheme, android:host, and android:path definitions.• Note exported components (android:exported="true") handling sensitive actions (e.g., login, pay, view salary).2️⃣ Use adb shell am start -a android.intent.action.VIEW -d "myapp://openSalaryReport" to invoke custom URLs without prior authentication.• Confirm if sensitive screens or actions trigger without restrictions.3️⃣ Test with tampered or malformed URL parameters.4️⃣ Attempt sending forged Intents using adb shell am startservice or am broadcast targeting exported components.5️⃣ Check for missing permission checks or validation inside the receiving activity or service.**iOS:**1️⃣ Extract the IPA and inspect **Info.plist** for CFBundleURLSchemes and CFBundleURLTypes.• Identify custom schemes like mybankapp://2️⃣ Use a browser or app to launch crafted URLs targeting sensitive app functions.• Confirm if app screens or actions trigger without authentication, session validation, or input checks.3️⃣ Test with manipulated parameters (e.g. mybankapp://pay?amount=10000&account=otheruser). |
| **Expected Result** | - Sensitive app functionality is never exposed via custom URL schemes unless protected.- If deep links exist for sensitive actions, they enforce:✔ Active session validation✔ Input validation✔ Signature/token check (if applicable)✔ Permission or authorization validation- Malformed or unauthorized URLs are rejected or safely handled. |
| **Actual Result** | (To be filled after testing — e.g. Unauthenticated custom URL triggered salary view screen directly) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if any sensitive functionality accessible via unauthenticated, unprotected URL schemes |
| **Evidence** | Example findings:• AndroidManifest.xml exports PayActivity with deep link myapp://payAmount without permission check.• iOS CFBundleURLTypes exposes paySalary action with no session validation.• Malformed URL myapp://open?token=abc123<script> processed without sanitization.• Sensitive screens accessed via adb shell am start -d "myapp://viewSalary" while logged out. |
| **Mitigation Recommendation** | - Remove custom URL handlers for sensitive functions unless strictly required.- Apply runtime checks to confirm:✔ Active, valid authenticated session✔ User authorization for action✔ Input validation and sanitization on all parameters✔ (Optionally) cryptographic signature verification for URL parameters- Set android:exported="false" for private components handling custom URLs on Android.- Validate and test deep link security in every release pentest.- Include safe deep link practices in mobile app secure coding standards. |

**Verifying Sensitive Functionality Not Exported via IPC Facilities Without Protection**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application does not export sensitive functionality through IPC mechanisms (Intents, BroadcastReceivers, Services, ContentProviders) unless explicitly required — and when exposed, these components are properly secured with permissions, signature validation, input validation, and session enforcement. |
| **Pre-conditions** | - APK build available- Device/emulator access- MobSF, JADX, ADB tools- Frida (optional) |
| **Test Data** | - Crafted Intents, broadcast messages, and ContentProvider requests targeting sensitive app components |
| **Test Steps** | **Android:**1️⃣ Decompile APK using **MobSF** or **JADX**.• Review AndroidManifest.xml for components with android:exported="true" or unspecified (defaults to true pre-Android 12).• Identify exported <activity>, <service>, <receiver>, and <provider> components.• Cross-check these against app’s feature scope — flag any sensitive or unnecessary exposed components.2️⃣ Use **adb shell am start**, **am startservice**, or **am broadcast** to craft and send test Intents to these exported components.• Example: adb shell am start -n com.example.app/.SensitiveActivity --es token "fake"3️⃣ Observe app behavior:• Confirm whether sensitive screens or actions trigger without authentication/session checks.• Test malformed or malicious extras (oversized strings, code injection attempts).4️⃣ For exported **BroadcastReceivers**:• Use adb shell am broadcast to send crafted broadcasts to receivers handling sensitive actions (e.g. logout, reset password).5️⃣ For **ContentProviders**:• Identify android:exported="true" providers.• Attempt unauthorized content:// queries and updates via adb or other apps.6️⃣ (Optional) Use **Frida/Objection** to hook component entry points and inspect IPC parameter handling and permission checks.7️⃣ Check MobSF scan report for exported component risks under IPC security section. |
| **Expected Result** | - No sensitive functionality (login, payment, salary reports, token resets, encryption keys, privileged APIs) is exposed through exported components unless necessary.- All exported components enforce permission checks (android:permission or runtime validation).- All IPC inputs validated and sanitized.- Unauthorized, malformed, or unprivileged IPC attempts are rejected. |
| **Actual Result** | (To be filled after testing — e.g. Exported PaySalaryService invoked via adb without permission, processed payment action) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if sensitive functionality is exposed via unprotected exported components |
| **Evidence** | Example findings:• AndroidManifest.xml exports SensitiveActivity with no permission protection.• ADB command launched sensitive Intent activity while logged out.• Exported BroadcastReceiver processed forged logout Intent.• ContentProvider exported publicly allowed unauthorized salary record query. |
| **Mitigation Recommendation** | - Set android:exported="false" for all private components.- For exported components, require strict permissions via android:permission attribute.- Validate all incoming IPC data for type, length, format, and business logic.- Implement runtime session and authentication checks on IPC entry points.- Remove or restrict unnecessary exported receivers and services.- Review IPC exposure in every release pentest and secure code review.- Include IPC hardening controls in secure coding guidelines. |

**Verifying JavaScript Is Disabled in WebViews Unless Explicitly Required**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application disables JavaScript execution in all embedded WebViews unless explicitly required by legitimate, secure business functionality — and if enabled, it is properly restricted and validated. |
| **Pre-conditions** | - APK/IPA production build available- MobSF, JADX (Android)- Hopper/class-dump (iOS)- Device/emulator access |
| **Test Data** | - Test URL hosting benign JavaScript payload (for validation, not attack)- Modified WebView content and configurations |
| **Test Steps** | **Android:**1️⃣ Decompile APK using **MobSF** or **JADX**.• Search for instances of WebView.• Check all WebSettings configurations:→ Look for webView.getSettings().setJavaScriptEnabled(true)2️⃣ Verify where JavaScript is enabled — confirm it's only within components that legitimately need it (e.g. a trusted payment page).3️⃣ Install app and open screens using WebViews.• Monitor with Burp Suite/mitmproxy (if traffic not pinned).• Load test HTML pages with JavaScript payloads and confirm execution.4️⃣ If enabled, confirm secure configurations:✔ No addJavascriptInterface() calls unless absolutely necessary (and if present, only to trusted interfaces).✔ Trusted, HTTPS-only content loaded.**iOS:**1️⃣ Extract IPA and review source code (if possible) or use **Hopper/class-dump**.• Check for WKPreferences.javaScriptEnabled = true or older UIWebView configs.2️⃣ Identify WebView instantiations and where JavaScript is enabled.3️⃣ Launch app and test screens rendering Web content — observe if JavaScript is executed.4️⃣ Confirm content source and security controls in place (CSP headers, TLS enforcement). |
| **Expected Result** | - JavaScript is disabled in all WebViews by default.- JavaScript enabled only where absolutely necessary, with validated, trusted content sources.- No addJavascriptInterface() unless safe and restricted.- HTTPS-only content enforced on enabled WebViews.- No unprotected JavaScript execution on arbitrary or external content. |
| **Actual Result** | (To be filled after testing — e.g. JavaScript enabled in WebView loading external salary data page without input validation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if JavaScript is enabled unnecessarily or insecurely in WebViews |
| **Evidence** | Example findings:• webView.getSettings().setJavaScriptEnabled(true) in SalaryActivity without URL validation.• WebView loads http://salary.example.com with active JavaScript.• addJavascriptInterface() used for salary operations without restrictions.• iOS WKPreferences.javaScriptEnabled set to true for untrusted web content. |
| **Mitigation Recommendation** | - Disable JavaScript in all WebViews by default using:✔ Android: webView.getSettings().setJavaScriptEnabled(false)✔ iOS: WKPreferences.javaScriptEnabled = false- Enable only if strictly necessary for trusted, validated content.- Enforce HTTPS-only loading on WebViews.- Avoid addJavascriptInterface() unless unavoidable — and use secure, restricted interfaces.- Perform input validation and implement CSP if using WebViews to load dynamic content.- Regularly review WebView configurations in code reviews and pentests. |

**Verifying WebView Protocol Handlers Are Restricted to Minimum Set (Only HTTPS)**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that all WebViews in the mobile application are configured to allow only the minimum necessary protocol handlers — ideally only https:// — and block dangerous handlers such as file://, tel://, mailto://, and custom app schemes unless explicitly required and secured. |
| **Pre-conditions** | - APK/IPA production build available- MobSF, JADX (Android)- Hopper/class-dump (iOS)- Device/emulator access |
| **Test Data** | - Custom HTML pages containing test hyperlinks to https://, http://, file://, tel://, mailto://, and intent:// URIs |
| **Test Steps** | **Android:**1️⃣ Decompile APK using **MobSF** or **JADX**.• Locate all WebView instances.• Check shouldOverrideUrlLoading() implementation in custom WebViewClient classes.• Confirm URL schemes are explicitly whitelisted (e.g. only https://).2️⃣ Create a test HTML page with multiple links using various URI schemes:html<br><a href="https://secure.example.com">Secure Link</a><br><a href="http://insecure.example.com">Insecure Link</a><br><a href="file:///etc/passwd">File Access</a><br><a href="tel:1234567890">Phone Call</a><br>3️⃣ Load this page into the app’s WebView and attempt to interact with each link.• Confirm only https:// links are allowed to proceed.• All other schemes should be blocked or handled safely.4️⃣ Check for the use of WebViewClient.shouldInterceptRequest() or shouldOverrideUrlLoading() overrides handling other protocols insecurely.**iOS:**1️⃣ Extract IPA and inspect source via Hopper/class-dump.• Locate WKNavigationDelegate implementations.• Review decidePolicyFor navigationAction methods.• Confirm URL scheme validation permits only https:// links.2️⃣ Load the same custom HTML page into the app’s WKWebView.• Interact with each link.• Confirm secure behavior:✔ Only https:// allowed✔ All others rejected or trigger safe error handlers |
| **Expected Result** | - Only https:// links are permitted to load within WebViews.- file://, tel://, mailto://, and other untrusted protocols are explicitly rejected.- No silent, unintended opening of local files, phone dialers, email clients, or external apps through WebView URIs.- WebViewClient and WKNavigationDelegate implementations enforce URI scheme allowlisting. |
| **Actual Result** | (To be filled after testing — e.g. WebView allowed file:// URI to load local content without restriction) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if any dangerous protocol handler is supported or allowed by default |
| **Evidence** | Example findings:• No shouldOverrideUrlLoading() override present in SecureWebViewClient — all URIs loaded without restriction.• WebView loaded file:///etc/passwd test link.• tel: link opened dialer from WebView without user interaction.• iOS decidePolicyFor navigationAction method missing scheme restrictions — allowed tel:// links. |
| **Mitigation Recommendation** | - Implement strict URL scheme validation in:✔ Android shouldOverrideUrlLoading() to allow only https://.✔ iOS decidePolicyFor navigationAction to enforce https:// only.- Disable JavaScript interfaces in WebViews by default.- Disallow file:// loading via webView.getSettings().setAllowFileAccess(false).- Avoid auto-handling tel://, mailto://, or other handlers unless explicitly needed — and always prompt the user.- Validate URI schemes against an approved allowlist before loading external links in WebViews.- Include this validation in mobile app code reviews, pentests, and QA acceptance criteria. |

**Verifying WebView Protocol Handlers Are Restricted to Minimum Set (Only HTTPS)**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that all WebViews in the mobile application are configured to allow only the minimum necessary protocol handlers — ideally only https:// — and block dangerous handlers such as file://, tel://, mailto://, and custom app schemes unless explicitly required and secured. |
| **Pre-conditions** | - APK/IPA production build available- MobSF, JADX (Android)- Hopper/class-dump (iOS)- Device/emulator access |
| **Test Data** | - Custom HTML pages containing test hyperlinks to https://, http://, file://, tel://, mailto://, and intent:// URIs |
| **Test Steps** | **Android:**1️⃣ Decompile APK using **MobSF** or **JADX**.• Locate all WebView instances.• Check shouldOverrideUrlLoading() implementation in custom WebViewClient classes.• Confirm URL schemes are explicitly whitelisted (e.g. only https://).2️⃣ Create a test HTML page with multiple links using various URI schemes:html<br><a href="https://secure.example.com">Secure Link</a><br><a href="http://insecure.example.com">Insecure Link</a><br><a href="file:///etc/passwd">File Access</a><br><a href="tel:1234567890">Phone Call</a><br>3️⃣ Load this page into the app’s WebView and attempt to interact with each link.• Confirm only https:// links are allowed to proceed.• All other schemes should be blocked or handled safely.4️⃣ Check for the use of WebViewClient.shouldInterceptRequest() or shouldOverrideUrlLoading() overrides handling other protocols insecurely.**iOS:**1️⃣ Extract IPA and inspect source via Hopper/class-dump.• Locate WKNavigationDelegate implementations.• Review decidePolicyFor navigationAction methods.• Confirm URL scheme validation permits only https:// links.2️⃣ Load the same custom HTML page into the app’s WKWebView.• Interact with each link.• Confirm secure behavior:✔ Only https:// allowed✔ All others rejected or trigger safe error handlers |
| **Expected Result** | - Only https:// links are permitted to load within WebViews.- file://, tel://, mailto://, and other untrusted protocols are explicitly rejected.- No silent, unintended opening of local files, phone dialers, email clients, or external apps through WebView URIs.- WebViewClient and WKNavigationDelegate implementations enforce URI scheme allowlisting. |
| **Actual Result** | (To be filled after testing — e.g. WebView allowed file:// URI to load local content without restriction) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if any dangerous protocol handler is supported or allowed by default |
| **Evidence** | Example findings:• No shouldOverrideUrlLoading() override present in SecureWebViewClient — all URIs loaded without restriction.• WebView loaded file:///etc/passwd test link.• tel: link opened dialer from WebView without user interaction.• iOS decidePolicyFor navigationAction method missing scheme restrictions — allowed tel:// links. |
| **Mitigation Recommendation** | - Implement strict URL scheme validation in:✔ Android shouldOverrideUrlLoading() to allow only https://.✔ iOS decidePolicyFor navigationAction to enforce https:// only.- Disable JavaScript interfaces in WebViews by default.- Disallow file:// loading via webView.getSettings().setAllowFileAccess(false).- Avoid auto-handling tel://, mailto://, or other handlers unless explicitly needed — and always prompt the user.- Validate URI schemes against an approved allowlist before loading external links in WebViews.- Include this validation in mobile app code reviews, pentests, and QA acceptance criteria. |

**Verifying WebView Native Method Exposure & Safe JavaScript Content Handling**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application’s WebViews do not expose unprotected native methods to JavaScript via addJavascriptInterface() (Android) or equivalent, and that WebViews only execute JavaScript either from trusted, validated HTTPS content or local app-packaged files — not untrusted or dynamic remote sources. |
| **Pre-conditions** | - APK/IPA build available- Device/emulator access- MobSF, JADX (Android)- Hopper/class-dump (iOS) |
| **Test Data** | - WebView test HTML pages with embedded JavaScript calling native interfaces- Modified local and remote HTML/JS content |
| **Test Steps** | **Android:**1️⃣ Decompile APK using **MobSF** or **JADX**.• Search for all calls to webView.addJavascriptInterface().• Identify which Java/Kotlin classes are being exposed to JavaScript and under what name.• Check for unsafe interfaces or sensitive methods (e.g., methods performing payments, token management, file access).2️⃣ Review WebView content loading calls:• Confirm all loaded URLs are either file:///android\_asset/ or https:// from a trusted domain.• Detect any dynamic, untrusted, or plaintext http:// content loading.3️⃣ If addJavascriptInterface() is used, confirm:• The WebView does **not load untrusted remote content**.• Only app-local JavaScript can invoke the native methods.4️⃣ Test the WebView runtime:• Load a test page with JavaScript attempting to call exposed native interfaces.• Confirm it only works when content is from a verified local file or trusted HTTPS page.**iOS:**1️⃣ Extract IPA and inspect source via Hopper/class-dump.• Locate WKWebView.configuration.userContentController.add() calls.• Identify message handlers registered for native access.• Review content loaded into WebViews.• Confirm only trusted, static HTML content or verified HTTPS URLs are loaded into WebViews running JavaScript.2️⃣ Test by attempting to load dynamic/remote or injected HTML content in the WebView and invoking registered message handlers — confirm these are blocked. |
| **Expected Result** | - No unprotected native methods exposed via addJavascriptInterface() unless explicitly required.- WebViews only execute JavaScript from app-local file:// assets or trusted, validated HTTPS sources.- No untrusted or dynamic remote JavaScript content can invoke native app functions.- Native interfaces are not bound to arbitrary WebView content.- No http:// content loading into JavaScript-enabled WebViews. |
| **Actual Result** | (To be filled after testing — e.g. SalaryWebView exposed paySalary() via addJavascriptInterface() with no restrictions, accessible from remote site) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if untrusted content can access native interfaces or JavaScript from external sources executed insecurely |
| **Evidence** | Example findings:• webView.addJavascriptInterface(new PaymentHandler(), "pay") exposed payment methods to JavaScript.• WebView loaded http:// remote page with JavaScript access.• No validation of JavaScript origin in shouldOverrideUrlLoading() or shouldInterceptRequest().• iOS WKUserContentController.add() exposed native handlers without verifying loaded content origin. |
| **Mitigation Recommendation** | - Avoid using addJavascriptInterface() unless strictly required.- If needed, expose only safe, non-sensitive methods and ensure WebView loads only from trusted local app assets or validated HTTPS domains.- Disable JavaScript for WebViews that don't explicitly need it (setJavaScriptEnabled(false)).- Implement strong validation in shouldOverrideUrlLoading() and shouldInterceptRequest() to block untrusted URIs and dynamic content.- For iOS, restrict WKWebView JavaScript message handlers to app-local packaged files or secure HTTPS content only.- Regularly review WebView configurations during secure code reviews and pentests. |
|  |  |

**Verifying Secure Object Deserialization**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that if object deserialization is implemented in the mobile application, it uses secure, safe serialization APIs — avoiding insecure mechanisms like ObjectInputStream.readObject() or unvalidated deserialization of attacker-controlled data. |
| **Pre-conditions** | - APK/IPA production build available- Device/emulator access- MobSF, JADX (Android)- Hopper/class-dump (iOS)- Frida (optional) |
| **Test Data** | - Crafted serialized payloads (if possible)- Malformed, oversized, or manipulated serialized objects via Intents, network APIs, IPC |
| **Test Steps** | **Android:**1️⃣ Decompile APK using **MobSF** or **JADX**.• Search for calls to ObjectInputStream.readObject(), readValue(), or similar native deserialization methods.• Check for use of custom object deserialization in Intent, Bundle, or IPC handlers.2️⃣ Review deserialization logic for validation:• Confirm only trusted, verified data is deserialized.• Ensure input validation occurs before deserialization.3️⃣ Review use of JSON/XML deserialization libraries:• If using Gson, check for new Gson().fromJson() calls — confirm no dynamic class types, custom deserializers validated.• If Jackson used, confirm enableDefaultTyping() is not enabled (as it’s dangerous).• Prefer safe JSON libraries like Moshi or Gson with TypeAdapters.4️⃣ Attempt runtime tampering via Frida or adb:• Send malformed or oversized serialized payloads via Intents, Bundles, or APIs.• Confirm app rejects, safely handles, or fails securely. |
| **Expected Result** | - No use of unsafe Java/Kotlin deserialization APIs (ObjectInputStream.readObject() on untrusted data).- All deserialization occurs through safe, schema-based, or validated libraries.- All deserialized input validated before use.- No untrusted or attacker-controlled serialized data processed without integrity or type enforcement. |
| **Actual Result** | (To be filled after testing — e.g. ObjectInputStream.readObject() used in IPC receiver without validation) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if insecure deserialization detected on untrusted or remotely supplied data |
| **Evidence** | Example findings:• ObjectInputStream.readObject() found in SalaryReceiver.java without type or permission check.• Gson used without TypeAdapter validation for deserialization of user data from API response.• Jackson enableDefaultTyping() enabled in salary API deserialization.• No input validation before deserialization of serialized Intent extras. |
| **Mitigation Recommendation** | - Replace all insecure deserialization code using ObjectInputStream.readObject() with safe, schema-based alternatives.- Use modern, secure JSON libraries like Gson (with TypeAdapter), Moshi, or Jackson (without polymorphic typing).- Implement strict input validation before any deserialization.- Avoid deserializing data from Intents, Bundles, or external sources unless properly signed and integrity-verified.- Include deserialization validation in secure code reviews and mobile pentests.- Test deserialization paths for DoS, injection, and object mismatch exploits. |

**Verifying Protection Against Screen Overlay Attacks**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application implements protections against screen overlay (tapjacking) attacks by detecting overlay windows and disabling input for sensitive actions, and by using secure window flags where necessary to block unauthorized overlays and screen captures. |
| **Pre-conditions** | - APK build available- Test device or emulator- Installed overlay-testing app (like Tapjacking demo apps, or use Frida/adb overlays) |
| **Test Data** | - Test overlay app showing transparent or disguised overlay over the target app’s sensitive screens |
| **Test Steps** | 1️⃣ Install both the target app and a test overlay app (or create one using SYSTEM\_ALERT\_WINDOW permission on Android 11 or earlier).2️⃣ Launch the overlay app to display a transparent window over the target app’s sensitive activities (login, payment, password input, permissions).3️⃣ Open the target app’s sensitive screens.• Attempt interactions (taps, inputs) with overlay active.• Observe whether the app detects overlays and disables input or blocks interactions.4️⃣ Check app source (MobSF/JADX) for implementation of:• Settings.canDrawOverlays() checks before sensitive actions.• Usage of FLAG\_SECURE on sensitive Activity windows (getWindow().setFlags(WindowManager.LayoutParams.FLAG\_SECURE, ...))5️⃣ Try taking screenshots while sensitive screen is active — verify if it’s blocked by FLAG\_SECURE.6️⃣ Observe whether the app warns the user or prevents actions if overlays are detected. |
| **Expected Result** | - App detects overlay windows during sensitive operations and disables inputs or alerts the user.- FLAG\_SECURE used on payment/password/token screens.- Unauthorized screen captures blocked.- No sensitive operations possible while an overlay is active. |
| **Actual Result** | (To be filled after testing — e.g. Salary approval screen allowed tap actions while overlay was active; no FLAG\_SECURE applied) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if sensitive operations remain unprotected against overlays |
| **Evidence** | Example findings:• No overlay detection in onWindowFocusChanged() or sensitive Activities.• getWindow().setFlags(FLAG\_SECURE) missing on payment/password screens.• Sensitive salary confirmation accepted tap actions while overlay was active.• Screenshots permitted on sensitive token generation screen. |
| **Mitigation Recommendation** | - Detect overlays using Settings.canDrawOverlays() or similar methods in sensitive activities.- Disable or postpone sensitive actions when overlays are detected.- Apply FLAG\_SECURE to window flags for all sensitive screens to block screen captures and overlays:java<br>getWindow().setFlags(WindowManager.LayoutParams.FLAG\_SECURE, WindowManager.LayoutParams.FLAG\_SECURE);<br>- Regularly review and test for tapjacking vulnerabilities during pentests and secure code reviews.- Educate users via UI alerts if overlays interfere with secure app operations. |

**Verifying Prevention of Third-Party Keyboard Usage on Sensitive Input Fields**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Confirm that the application prevents or restricts the use of custom third-party keyboards whenever sensitive data (passwords, PINs, payment tokens, salary data) is being entered in the application. |
| **Pre-conditions** | - APK/IPA production build available- Test device with third-party keyboards installed and enabled (e.g. Gboard, SwiftKey, custom test keyboard) |
| **Test Data** | - Password, PIN, token, salary input fields on login, payment, and sensitive forms |
| **Test Steps** | **Android:**1️⃣ Install a third-party keyboard app on the test device.2️⃣ Open the target app and navigate to sensitive input screens (password, PIN, salary entry).3️⃣ Tap into the input fields — verify which keyboard is invoked.4️⃣ Check if the app forces the use of the system keyboard or blocks custom keyboards on these fields.5️⃣ Decompile the APK using **MobSF/JADX** and check for:• Usage of setPrivateImeOptions("nm") or similar for secure fields.• Application of FLAG\_SECURE to sensitive activities.6️⃣ Review input fields' XML layouts and programmatic configurations for secure input types (InputType.TYPE\_TEXT\_VARIATION\_PASSWORD, etc).**iOS:**1️⃣ Install a third-party keyboard app on the test device.2️⃣ Open the app and navigate to sensitive input fields.3️⃣ Tap into the fields — verify whether third-party keyboards are allowed.4️⃣ Decompile IPA or use class-dump/Hopper to review UITextField/UITextView configurations:• Check if secureTextEntry = YES is set.• Confirm whether textField.inputAssistantItem.leadingBarButtonGroups = [] or third-party keyboard restrictions applied.5️⃣ Review App Transport Security settings for secure keyboard interactions (if applicable). |
| **Expected Result** | - On all sensitive input fields, only the system default keyboard should appear.- No third-party keyboards should be allowed when entering sensitive data.- Secure input flags and input types applied to sensitive fields.- On iOS, secureTextEntry enabled and third-party keyboards restricted. |
| **Actual Result** | (To be filled post-test — e.g. Third-party keyboard allowed input on salary PIN field, no restrictions applied) |
| **Status** | (Pass / Fail Detected) |
| **Severity** | **High** if third-party keyboard allowed on sensitive fields |
| **Evidence** | Example findings:• Password field permitted Gboard and SwiftKey on Android.• Salary token input allowed third-party keyboard on iOS.• No FLAG\_SECURE or secure input type configured on sensitive screens.• secureTextEntry flag missing on UITextField for password inputs. |
| **Mitigation Recommendation** | - On **Android**:• Apply FLAG\_SECURE on sensitive screens:java<br>getWindow().setFlags(WindowManager.LayoutParams.FLAG\_SECURE, WindowManager.LayoutParams.FLAG\_SECURE);<br>• Use secure input types: InputType.TYPE\_TEXT\_VARIATION\_PASSWORD, TYPE\_NUMBER\_VARIATION\_PASSWORD, etc.• Avoid exposing sensitive data to custom IMEs.- On **iOS**:• Set secureTextEntry = YES for sensitive UITextField/UITextView.• Disable third-party keyboards where necessary via:swift<br>override func textFieldShouldBeginEditing(\_ textField: UITextField) -> Bool {<br> textField.inputAssistantItem.leadingBarButtonGroups = []<br> return true<br>}<br>- Regularly test and review secure keyboard configurations in mobile pentests and code audits. |

**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. |