**Activity Hijacking**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Detect if exported activities are improperly exposed and can be invoked by unauthorized apps. |
| **Pre-conditions** | MobSF installed. APK available. adb or app simulator for sending Intents. |
| **Test Data** | APK: com.example.app\_1.0.0.apk |
| **Test Steps** | 1. Upload APK to MobSF. 2. Review Components for exported activities. 3. Use adb to start activity externally: adb shell am start -n com.example.app/.DebugActivity |
| **Expected Result** | Activities should block unauthorized external launches without proper permissions. |
| **Actual Result** | **DebugActivity** and **TestActivity** launchable by third-party apps without restrictions. |
| **Status** | Fail Detected |
| **Severity** | High |
| **Evidence** | Command: adb shell am start -n com.example.app/.DebugActivity **Result:** Activity launches without authentication. |
| **Mitigation Recommendation** | Set exported="false" for internal-only activities. Apply intent filters and permissions for any exported activity. Remove test/debug activities from production builds. |

**Service Abuse**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Identify if exported services can be started or bound to by unauthorized apps. |
| **Pre-conditions** | MobSF installed. APK file available. adb or app simulator for intent-based service interaction. |
| **Test Data** | APK: com.example.app\_1.0.0.apk |
| **Test Steps** | 1. Upload APK to MobSF. 2. Locate exported services. 3. Attempt to start service via adb: adb shell am startservice -n com.example.app/.TestService |
| **Expected Result** | Exported services should enforce permission checks or deny unauthorized requests. |
| **Actual Result** | **TestService** starts successfully without any restrictions or validation. |
| **Status** | Fail Detected |
| **Severity** | High |
| **Evidence** | Command: adb shell am startservice -n com.example.app/.TestService **Result:** Service starts without restrictions. |
| **Mitigation Recommendation** | Set exported="false" for internal services. Apply custom permissions to control external access. Validate incoming Intents within the service code. |

**Intent Spoofing**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Check if the app improperly trusts external Intents without validating their origin or content. |
| **Pre-conditions** | MobSF installed. APK file available. adb or intent-crafting tool. |
| **Test Data** | APK: com.example.app\_1.0.0.apk Malicious payload: malicious\_data |
| **Test Steps** | 1. Upload APK to MobSF. 2. Identify exported components. 3. Send a crafted Intent: adb shell am start -n com.example.app/.DebugActivity --es "payload" "malicious\_data" |
| **Expected Result** | The app should validate the Intent’s origin and sanitize data before use. |
| **Actual Result** | **DebugActivity** processes and displays the malicious\_data without validation. |
| **Status** | Fail Detected |
| **Severity** | Medium-High |
| **Evidence** | Command: adb shell am start -n com.example.app/.DebugActivity --es "payload" "malicious\_data" **Result:** App processes injected data unsafely. |
| **Mitigation Recommendation** | Always verify the origin of Intents. Apply permission and intent-filter constraints. Sanitize and validate incoming Intent data before processing. |

**Business Logic Bypass**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Identify if the backend enforces proper business logic rules independently of client-side restrictions, preventing unauthorized operations through API request tampering. |
| **Pre-conditions** | - Intercepting proxy (e.g. Burp Suite) configured. - Mobile application installed and functional. - Test user account credentials. - APK analyzed for relevant endpoints (optional via MobSF). |
| **Test Data** | - APK: com.example.app\_1.0.0.apk - Test user account - Modified API requests violating business rules (e.g. exceeding allowed transaction limits, or submitting invalid workflow states). |
| **Test Steps** | 1. Intercept app traffic while performing actions with the mobile app. 2. Identify API requests where business rules are enforced on the client (e.g. disabling a button, or restricting input fields). 3. Modify the intercepted API request (e.g. increase payment amount or change transaction status directly). 4. Send the modified request and observe backend response. |
| **Expected Result** | The backend should reject the request, enforcing the same business logic rules server-side as on the client. |
| **Actual Result** | The server processed the unauthorized request, bypassing business rules enforced only by the mobile app. |
| **Status** | Fail Detected |
| **Severity** | High |
| **Evidence** | **Original Request:** http POST /api/transfer HTTP/1.1 Host: api.example.com Authorization: Bearer [token] Content-Type: application/json { "amount": 5000 } **Modified Request (exceeding allowed client-side limit of 1000):** http POST /api/transfer HTTP/1.1 Host: api.example.com Authorization: Bearer [token] Content-Type: application/json { "amount": 100000 } **Response:** http HTTP/1.1 200 OK { "status": "success" } |
| **Mitigation Recommendation** | - Enforce all business rules (limits, workflow states, etc.) on the backend. - Reject any requests violating server-side logic, regardless of client behavior. - Remove reliance on client-side enforcement for critical logic decisions. - Perform regular API security assessments for business logic flaws. |

**Rate limit Bypass**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Verify that server-side rate limiting and anti-automation controls are enforced for mobile API endpoints to prevent abuse through repeated, automated requests. |
| **Pre-conditions** | - Intercepting proxy (e.g. Burp Suite Intruder or Turbo Intruder). - Mobile application installed. - Valid test account credentials. - API endpoints identified (via MobSF or proxy interception). |
| **Test Data** | - APK: com.example.app\_1.0.0.apk - Valid credentials. - Automated or rapid repeated API requests. |
| **Test Steps** | 1. Identify rate-limited or sensitive endpoints (e.g. login, OTP requests, or balance checks). 2. Use Intruder or custom scripts to send a large number of requests in rapid succession. 3. Monitor the server’s response behavior for throttling, blocking, or error messages. |
| **Expected Result** | The server should enforce rate limiting (e.g. 429 Too Many Requests) and temporary IP/account blocking after threshold violations. |
| **Actual Result** | No rate limiting detected — the API allowed unlimited repeated requests without restrictions or delays. |
| **Status** | Fail Detected |
| **Severity** | Medium-High |
| **Evidence** | **Burp Intruder Results:** 500 login attempts sent within 60 seconds — all received valid 200 OK or 401 Unauthorized responses without throttling or blocking. No HTTP 429 or anti-automation defenses triggered. |
| **Mitigation Recommendation** | - Implement proper server-side rate limiting for sensitive endpoints. - Return HTTP 429 status codes on threshold breaches. - Temporarily block IP addresses or user accounts after multiple failed attempts. - Consider device fingerprinting and CAPTCHA after repeated actions. |

**Broken authorization in Mobile API**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Determine whether API endpoints properly enforce server-side authorization checks to prevent unauthorized access to resources belonging to other users. |
| **Pre-conditions** | - Mobile application installed and running. - Intercepting proxy configured. - At least two valid user accounts (test users). - API endpoints identified via MobSF or traffic interception. |
| **Test Data** | - APK: com.example.app\_1.0.0.apk - Two sets of test credentials (User A & User B). - Modified API requests attempting to access resources belonging to other users. |
| **Test Steps** | 1. Log in as User A and capture an API request fetching user-specific data (e.g. profile info or orders). 2. Modify the intercepted request by replacing User A’s identifier with User B’s identifier. 3. Send the modified request and observe server behavior. |
| **Expected Result** | The server should enforce authorization checks and deny access to unauthorized resources with a 403 Forbidden or similar response. |
| **Actual Result** | The server returned sensitive data for the unauthorized user (User B), bypassing access controls. |
| **Status** | Fail Detected |
| **Severity** | High |
| **Evidence** | **Request as User A:** http POST /api/get\_user\_info HTTP/1.1 Host: api.example.com Authorization: Bearer token\_userA Content-Type: application/json { "user\_id": "A001" } **Modified Request (targeting User B):** http POST /api/get\_user\_info HTTP/1.1 Host: api.example.com Authorization: Bearer token\_userA Content-Type: application/json { "user\_id": "B001" } **Response:** http HTTP/1.1 200 OK { "user\_name": "victim\_user", "email": "victim@example.com" } |
| **Mitigation Recommendation** | - Enforce strict server-side authorization checks on every API request. - Use token-based identity mapping rather than relying on client-supplied identifiers. - Return HTTP 403 Forbidden for unauthorized resource access. - Regularly test APIs for IDOR (Insecure Direct Object Reference) and broken authorization vulnerabilities. |

**Business Logic Bypass via API Manipulation**

| **Field** | **Content** |
| --- | --- |
| **Test Objective** | Identify if the backend enforces proper business logic rules independently of client-side restrictions, preventing unauthorized operations through API request tampering. |
| **Pre-conditions** | - Intercepting proxy (e.g. Burp Suite) configured. - Mobile application installed and functional. - Test user account credentials. - APK analyzed for relevant endpoints (optional via MobSF). |
| **Test Data** | - APK: com.example.app\_1.0.0.apk - Test user account - Modified API requests violating business rules (e.g. exceeding allowed transaction limits, or submitting invalid workflow states). |
| **Test Steps** | 1. Intercept app traffic while performing actions with the mobile app. 2. Identify API requests where business rules are enforced on the client (e.g. disabling a button, or restricting input fields). 3. Modify the intercepted API request (e.g. increase payment amount or change transaction status directly). 4. Send the modified request and observe backend response. |
| **Expected Result** | The backend should reject the request, enforcing the same business logic rules server-side as on the client. |
| **Actual Result** | The server processed the unauthorized request, bypassing business rules enforced only by the mobile app. |
| **Status** | Fail Detected |
| **Severity** | High |
| **Evidence** | **Original Request:** http POST /api/transfer HTTP/1.1 Host: api.example.com Authorization: Bearer [token] Content-Type: application/json { "amount": 5000 } **Modified Request (exceeding allowed client-side limit of 1000):** http POST /api/transfer HTTP/1.1 Host: api.example.com Authorization: Bearer [token] Content-Type: application/json { "amount": 100000 } **Response:** http HTTP/1.1 200 OK { "status": "success" } |
| **Mitigation Recommendation** | - Enforce all business rules (limits, workflow states, etc.) on the backend. - Reject any requests violating server-side logic, regardless of client behavior. - Remove reliance on client-side enforcement for critical logic decisions. - Perform regular API security assessments for business logic flaws. |

**Mobile Application Architecture Security Model [Full Security Concern matrix]**

| **Component** | **Description** | **Possible Security Concerns (Comprehensive)** |
| --- | --- | --- |
| **Mobile Client (APK/IPA)** | The mobile app installed on the user's device. | - Debuggable flag enabled in production build- Insecure exported components (Activities, Services, Broadcast Receivers, Content Providers)- Hardcoded secrets, API keys, tokens- Insecure local data storage (SharedPreferences, SQLite, files)- Insecure cryptography (weak algorithms, improper key management)- SSL/TLS misconfiguration or absence of SSL Pinning- Insecure WebView usage (JavaScript enabled, no domain whitelisting)- Unprotected IPC mechanisms (Intents, AIDL services)- Missing root/jailbreak detection- Insecure logging (sensitive data in logs)- Reverse engineering protections absent |
| **API Server / Backend** | Main API and business logic server. | - Broken authentication (no token validation, session management issues)- Broken authorization (IDOR, missing RBAC)- Business logic bypass (enforced only client-side)- No server-side input validation- SQL/NoSQL/Command injections- Missing rate limiting and anti-automation- Insecure password reset mechanisms- Insecure file upload validation- Missing or weak error handling (stack traces, sensitive info in errors)- No CSRF protection (if applicable)- No audit logging or insufficient monitoring |
| **Authentication Server** | OAuth2/OIDC service for token issuance and user management. | - Weak password policies- Insecure token generation or predictable tokens- Missing refresh token implementation- Missing token revocation endpoint- No multi-factor authentication (MFA)- Tokens with excessive lifespan or no expiration- Improper handling of social logins (OAuth misconfigurations)- Lack of account lockout mechanisms |
| **Third-Party APIs / Services** | Payment gateways, SMS providers, analytics, ads, etc. | - Excessive privileges or over-permissioned tokens- No certificate pinning for third-party services- Hardcoded API credentials- Unvalidated or untrusted third-party responses- Data leakage via third-party SDK telemetry- Usage of insecure or deprecated SDKs |
| **Content Delivery Network (CDN)** | Distributes static files like images, JS, CSS. | - Insecure cache control headers (caching sensitive content)- CDN misconfigurations exposing sensitive resources- Serving outdated or vulnerable third-party libraries- No HTTPS enforcement on CDN URLs |
| **Push Notification Service** | Service for sending remote notifications to devices. | - Push tokens leaked or mismanaged- No authorization when sending push messages- Sensitive data in notification payloads- Unencrypted transport for push payloads |
| **Cloud Storage (e.g. S3)** | Cloud storage for media, backups, or documents. | - Public read/write access on buckets- No encryption at rest or in transit- Lack of file type restrictions- No antivirus or malware scanning on uploads- Weak or misconfigured bucket permissions- Exposed backup archives or database dumps |
| **App Store Metadata & Distribution** | App details, release config, permissions. | - Over-permissive declared permissions- Debug/test builds uploaded to production- Incomplete or inaccurate privacy policies- Insufficient disclosure of data collection practices |
| **Network Communication (Transport Layer)** | All data exchanged between app and services. | - No TLS (HTTP in use)- Use of deprecated TLS versions (SSL/TLS 1.0/1.1)- Lack of TLS certificate pinning- Weak cipher suites in use- No transport encryption for third-party calls- Sensitive data transmitted in cleartext |
| **In-App Payment Mechanism** | Handles purchases, wallets, or payment gateway interactions. | - Insecure client-side payment validation- Hardcoded payment tokens or API keys- No server-side validation for payment confirmations- Lack of tampering detection for transactions- No PCI DSS compliance checks where applicable |
| **Device Integrity & Root/Jailbreak Detection** | Detect compromised devices and prevent operation on them. | - Absence of root/jailbreak detection checks- Insecure bypassable detection mechanisms- Incomplete checks (not covering emulators, tampered system files, known binaries) |
| **Analytics / Crash Reporting SDKs** | Collect user data and app telemetry. | - Excessive data collection violating privacy policies- Unencrypted transmission of analytics data- Exposing PII (Personally Identifiable Information) in logs- No user opt-out mechanism |
| **Application Logs (Device & Remote)** | Logs generated locally or sent to remote servers. | - Sensitive data (passwords, tokens, card numbers) logged in plaintext- Verbose logs in production builds- Log files accessible to unauthorized apps (Android: world-readable files)- Logs exposing stack traces or internal errors |

| **Component** | **Possible Security Concerns** | **Recommended Mitigation** |
| --- | --- | --- |
| **Mobile Client (APK/IPA)** | - Debuggable flag enabled in production- Exposed exported components (Activities, Services, Broadcast Receivers, Content Providers)- Hardcoded secrets, API keys- Insecure local data storage- Weak or broken cryptography- SSL/TLS misconfigurations- Insecure WebView usage- Unprotected IPC mechanisms- Missing root/jailbreak detection- Sensitive data in logs- No reverse engineering protections | - Disable android:debuggable in production builds- Set exported="false" on internal components- Remove secrets, API keys from client; use secure backend APIs- Store sensitive data in Android Keystore / iOS Keychain, encrypt at rest- Use secure cryptography standards (AES-256, PBKDF2, Argon2)- Enforce TLS 1.2+ with certificate pinning- Use secure WebView configurations, disable JavaScript if not needed- Validate and restrict incoming IPC data- Implement root/jailbreak detection and enforce app termination if detected- Avoid logging sensitive data in production- Apply code obfuscation (ProGuard/R8, DexGuard, iOS equivalent) |
| **API Server / Backend** | - Broken authentication/authorization- IDOR vulnerabilities- Business logic bypass- No server-side validation- Injections (SQL/NoSQL/Command)- No rate limiting- Insecure password resets- Insecure file upload handling- Missing CSRF protection (if applicable)- Detailed error messages- No audit logs | - Enforce OAuth2/OIDC standards for authentication- Implement strict RBAC / ABAC for authorization- Always validate object ownership on server-side- Enforce all business logic on backend, never rely on client checks- Implement input validation and output encoding- Apply proper rate limiting (429 errors)- Secure password reset workflows with token-based verification and expiry- Validate file uploads: restrict types, size, scan for malware- Implement CSRF tokens for session-based APIs (if used)- Return generic error messages, log details server-side- Maintain audit logs of sensitive events |
| **Authentication Server** | - Weak password policy- Predictable or insecure tokens- No refresh token / revocation support- No MFA- Excessive token lifespan- OAuth misconfigurations- No account lockout mechanisms | - Enforce strong password policies (length, complexity, history)- Use securely generated random tokens (UUIDv4, crypto libraries)- Implement refresh token and token revocation endpoints- Support multi-factor authentication (OTP, device-bound tokens)- Set short lifespans for access tokens, long for refresh tokens- Follow secure OAuth 2.0/OIDC implementation guidelines (PKCE, state checks)- Enforce account lockout or rate limit failed login attempts |
| **Third-Party APIs / Services** | - Over-permissive API keys / tokens- No certificate pinning- Hardcoded API credentials- Unvalidated responses- Telemetry data leakage- Insecure SDK versions | - Follow least privilege principle for API scopes- Use TLS pinning for third-party connections where feasible- Remove hardcoded API keys; use secure token exchange- Validate all third-party API responses- Limit telemetry data to non-sensitive info- Keep SDKs updated; avoid deprecated or insecure libraries |
| **Content Delivery Network (CDN)** | - Caching sensitive content- Exposed sensitive resources- Outdated libraries- No HTTPS enforcement | - Set Cache-Control headers for sensitive content (no-store, no-cache)- Restrict public CDN resources, use signed URLs for private content- Keep third-party libraries updated- Enforce HTTPS/TLS on CDN endpoints |
| **Push Notification Service** | - Leaked push tokens- Unauthorized notifications- Sensitive data in payloads- Unencrypted transport | - Securely manage and rotate push tokens- Authenticate requests to push APIs- Avoid sending sensitive data in notifications- Use TLS for push payload delivery |
| **Cloud Storage (S3 / Azure / GCS)** | - Publicly readable buckets- No encryption at rest/in transit- Unrestricted file uploads- No malware scanning- Misconfigured permissions | - Enforce strict bucket IAM policies (private by default)- Enable encryption at rest (AES-256) and TLS for transfers- Implement file type, size, and content validation on upload- Integrate antivirus or malware scanning for uploads- Regularly audit and review storage permissions |
| **App Store Metadata & Distribution** | - Over-permissive declared permissions- Debug builds in production- Incomplete privacy policies- Undisclosed data collection | - Minimize requested permissions- Validate release build settings (non-debuggable, obfuscated)- Provide accurate and transparent privacy policies- Disclose all data collection practices and allow user consent |
| **Network Communication (Transport Layer)** | - No TLS- Weak or outdated TLS versions- No certificate pinning- Weak cipher suites- Cleartext sensitive data | - Enforce TLS 1.2+ and prefer 1.3- Disable SSLv3, TLS 1.0/1.1- Implement SSL/TLS certificate pinning- Use strong cipher suites (forward secrecy enabled)- Avoid transmitting sensitive data without encryption |
| **In-App Payment Mechanism** | - Client-side payment validation- Hardcoded tokens / keys- No server-side payment validation- No transaction tampering detection- PCI DSS non-compliance | - Validate all transactions server-side before finalizing- Remove payment keys from mobile clients- Implement HMAC signing for transaction integrity- Regularly review PCI DSS compliance for payment features |
| **Device Integrity & Root/Jailbreak Detection** | - No detection mechanisms- Bypassable or incomplete checks | - Implement runtime root/jailbreak detection- Terminate or restrict app on compromised devices- Use multiple detection techniques (files, binaries, system APIs, emulator indicators) |
| **Analytics / Crash Reporting SDKs** | - Excessive or sensitive data collection- Unencrypted telemetry- PII in logs- No user opt-out | - Limit telemetry to non-PII data- Ensure telemetry uses TLS for transmission- Avoid logging PII in crash reports- Provide users with data collection opt-out options |
| **Application Logs (Device & Remote)** | - Logging sensitive data- Verbose production logs- Logs readable by unauthorized apps- Exposing internal errors or stack traces | - Avoid logging passwords, tokens, or sensitive data- Disable verbose/debug logging in production- Protect log files with appropriate permissions- Replace internal error details with generic messages |

**Full mobile architecture security list**

| **Component** | **Possible Security Concerns** | **Recommended Mitigation** |
| --- | --- | --- |
| **Mobile Client (APK/IPA)** | - Debuggable flag enabled in production- Exposed exported components (Activities, Services, Broadcast Receivers, Content Providers)- Hardcoded secrets, API keys- Insecure local data storage- Weak or broken cryptography- SSL/TLS misconfigurations- Insecure WebView usage- Unprotected IPC mechanisms- Missing root/jailbreak detection- Sensitive data in logs- No reverse engineering protections | - Disable android:debuggable in production builds- Set exported="false" on internal components- Remove secrets, API keys from client; use secure backend APIs- Store sensitive data in Android Keystore / iOS Keychain, encrypt at rest- Use secure cryptography standards (AES-256, PBKDF2, Argon2)- Enforce TLS 1.2+ with certificate pinning- Use secure WebView configurations, disable JavaScript if not needed- Validate and restrict incoming IPC data- Implement root/jailbreak detection and enforce app termination if detected- Avoid logging sensitive data in production- Apply code obfuscation (ProGuard/R8, DexGuard, iOS equivalent) |
| **API Server / Backend** | - Broken authentication/authorization- IDOR vulnerabilities- Business logic bypass- No server-side validation- Injections (SQL/NoSQL/Command)- No rate limiting- Insecure password resets- Insecure file upload handling- Missing CSRF protection (if applicable)- Detailed error messages- No audit logs | - Enforce OAuth2/OIDC standards for authentication- Implement strict RBAC / ABAC for authorization- Always validate object ownership on server-side- Enforce all business logic on backend, never rely on client checks- Implement input validation and output encoding- Apply proper rate limiting (429 errors)- Secure password reset workflows with token-based verification and expiry- Validate file uploads: restrict types, size, scan for malware- Implement CSRF tokens for session-based APIs (if used)- Return generic error messages, log details server-side- Maintain audit logs of sensitive events |
| **Authentication Server** | - Weak password policy- Predictable or insecure tokens- No refresh token / revocation support- No MFA- Excessive token lifespan- OAuth misconfigurations- No account lockout mechanisms | - Enforce strong password policies (length, complexity, history)- Use securely generated random tokens (UUIDv4, crypto libraries)- Implement refresh token and token revocation endpoints- Support multi-factor authentication (OTP, device-bound tokens)- Set short lifespans for access tokens, long for refresh tokens- Follow secure OAuth 2.0/OIDC implementation guidelines (PKCE, state checks)- Enforce account lockout or rate limit failed login attempts |
| **Third-Party APIs / Services** | - Over-permissive API keys / tokens- No certificate pinning- Hardcoded API credentials- Unvalidated responses- Telemetry data leakage- Insecure SDK versions | - Follow least privilege principle for API scopes- Use TLS pinning for third-party connections where feasible- Remove hardcoded API keys; use secure token exchange- Validate all third-party API responses- Limit telemetry data to non-sensitive info- Keep SDKs updated; avoid deprecated or insecure libraries |
| **Content Delivery Network (CDN)** | - Caching sensitive content- Exposed sensitive resources- Outdated libraries- No HTTPS enforcement | - Set Cache-Control headers for sensitive content (no-store, no-cache)- Restrict public CDN resources, use signed URLs for private content- Keep third-party libraries updated- Enforce HTTPS/TLS on CDN endpoints |
| **Push Notification Service** | - Leaked push tokens- Unauthorized notifications- Sensitive data in payloads- Unencrypted transport | - Securely manage and rotate push tokens- Authenticate requests to push APIs- Avoid sending sensitive data in notifications- Use TLS for push payload delivery |
| **Cloud Storage (S3 / Azure / GCS)** | - Publicly readable buckets- No encryption at rest/in transit- Unrestricted file uploads- No malware scanning- Misconfigured permissions | - Enforce strict bucket IAM policies (private by default)- Enable encryption at rest (AES-256) and TLS for transfers- Implement file type, size, and content validation on upload- Integrate antivirus or malware scanning for uploads- Regularly audit and review storage permissions |
| **App Store Metadata & Distribution** | - Over-permissive declared permissions- Debug builds in production- Incomplete privacy policies- Undisclosed data collection | - Minimize requested permissions- Validate release build settings (non-debuggable, obfuscated)- Provide accurate and transparent privacy policies- Disclose all data collection practices and allow user consent |
| **Network Communication (Transport Layer)** | - No TLS- Weak or outdated TLS versions- No certificate pinning- Weak cipher suites- Cleartext sensitive data | - Enforce TLS 1.2+ and prefer 1.3- Disable SSLv3, TLS 1.0/1.1- Implement SSL/TLS certificate pinning- Use strong cipher suites (forward secrecy enabled)- Avoid transmitting sensitive data without encryption |
| **In-App Payment Mechanism** | - Client-side payment validation- Hardcoded tokens / keys- No server-side payment validation- No transaction tampering detection- PCI DSS non-compliance | - Validate all transactions server-side before finalizing- Remove payment keys from mobile clients- Implement HMAC signing for transaction integrity- Regularly review PCI DSS compliance for payment features |
| **Device Integrity & Root/Jailbreak Detection** | - No detection mechanisms- Bypassable or incomplete checks | - Implement runtime root/jailbreak detection- Terminate or restrict app on compromised devices- Use multiple detection techniques (files, binaries, system APIs, emulator indicators) |
| **Analytics / Crash Reporting SDKs** | - Excessive or sensitive data collection- Unencrypted telemetry- PII in logs- No user opt-out | - Limit telemetry to non-PII data- Ensure telemetry uses TLS for transmission- Avoid logging PII in crash reports- Provide users with data collection opt-out options |
| **Application Logs (Device & Remote)** | - Logging sensitive data- Verbose production logs- Logs readable by unauthorized apps- Exposing internal errors or stack traces | - Avoid logging passwords, tokens, or sensitive data- Disable verbose/debug logging in production- Protect log files with appropriate permissions- Replace internal error details with generic messages |