

行動應用 App 資安標章全攻略

「需求、開發到實驗室檢測的合規一體化實務」



L1



L2



L3

執行單位： 台北市電腦公會

單元三 | 基準說明篇

資安基準檢測實務與實驗室審查要領

將「檢測基準」化為「開發規格」，翻轉「事後被動受測」思維

實踐開發即合規的一體化實務

實驗室報告簽署人
鄭奕暘

課程地圖



單元三 | 基準說明篇 - 從「理解」到「內化」為開發習慣

深度剖析檢測基準的檢測項目、常見不合格案例，快速掌握送測前自我評估重點

三單元課程地圖

單元一

制度推動篇

打造「全面性防禦」管理力

制度緣起、架構、標章與市場價值

單元二

規範解析篇

建構「開發即合規」防護力

Security by Design，把規範化為開發控制點

單元三

本單元

基準說明篇

落實「實驗室實戰」檢測實務

檢測項目、常見盲點與送驗前 Checklist



課程主軸 從「外部檢測」→ 到「開發即合規」 以主動防禦貫穿 App 研發全生命週期

單元導讀 I

誰該觀看？為什麼？

輔助理解



公部門與需求單位

明確了解驗收標準與防禦基本要求
確保委外檢測、開發之資安合規性

推薦



App 開發商

透過送測自評 Checklist 審視開發缺陷
逐步將合規標準內化為開發習慣

推薦



檢測實驗室

了解基準防禦精神與判定盲點
提升檢測一致性與專業度

※ 杜絕開發商反覆修補、實驗室重測耗時與採購端驗收無期的惡性循環，以共通標準開啟『開發即合規』的三方共好新局。

單元導讀 II

學習目標

核心目標 I



了解 L1 至 L3 分級差異與適用情景，依據不同的功能特性，選擇適當的分類等級，避免過度或不足的常見誤區。

核心目標 II



了解行動應用 App 基本資安檢測基準的防護精神，建立風險導向為核心的思維框架，掌握合規背後的開發邏輯。

核心目標 III



洞悉實驗室靜態分析與動態分析的檢測原理，了解審查流程並掌握送測前之自我評估表重點，迅速提升首次通過率。

※ 本單元旨在對齊開發端與檢測端的標準語言，透過理解實驗室審查重點，幫助開發者逐漸內化為開發習慣。

PART 1

檢測基準架構與防護目標

從「檢測基準」反推「安全開發」的設計邏輯



思維翻轉

核心觀念：從「被動受測」到「資安左移」

❌ 舊思維：事後檢測與補救

開發完成後才送交實驗室檢測。以致發現安全漏洞後即面臨架構重寫，導致專案時程延宕、增加開發成本。



✅ 新思維：資安左移 (Shift Left)

將檢測基準納入需求規格書。從開發階段即防堵已知漏洞，將檢測做為合規性驗證，達到首次送測即合格。

檢測基準分類架構

如何判斷 App 適用的類別？

L1

無須使用者身分鑑別

不需要登入
即可使用主要功能

L2

須使用者身分鑑別

具備登入功能，如帳號
密碼、OTP及生物辨識等

L3

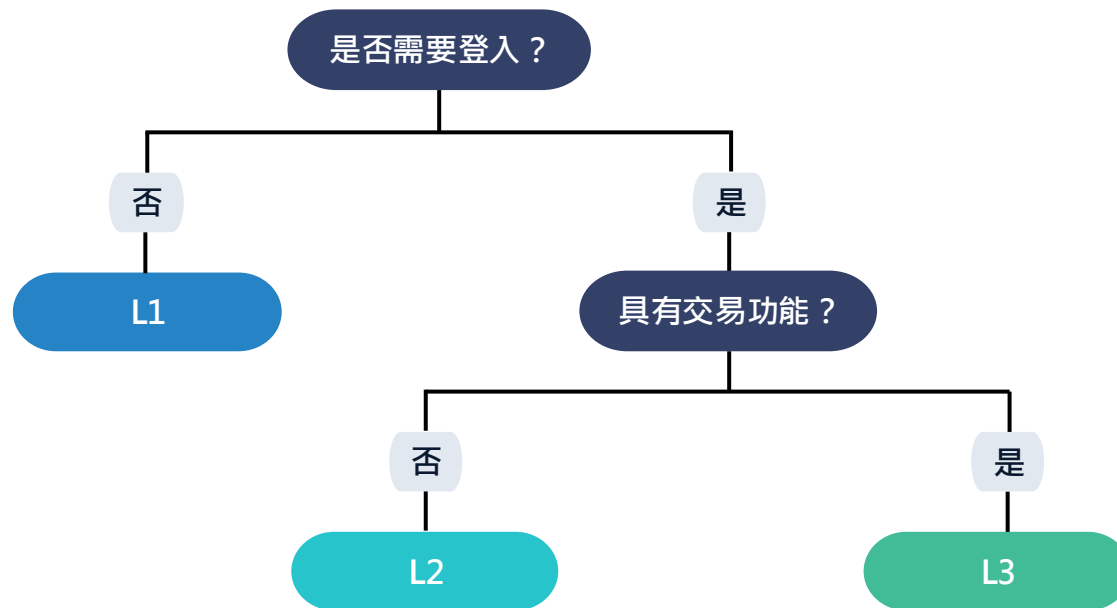
含有交易行為

包含但不限於付款、
轉帳及下單等交易功能

F

高安全性需求（選測）

針對安全防護較高的App
可視需求選擇是否加測



檢測基準防護目標 I

L1：建立 App 基本安全，避免資料洩漏與一般常見攻擊

1

程式發布安全：

- ◆ 明確說明欲存取之敏感性資料、行動裝置資源及宣告權限之用途

2

敏感性資料保護：

- ◆ 必須取得同意並提供拒絕之權利
- ◆ 避免使用弱加密及不安全之連線
- ◆ 避免洩漏於系統日誌及硬編碼

3

連線管理安全：

- ◆ 確保憑證有效且由信任之CA簽發
- ◆ 確實掌握封包流向 (國家與網域)

4

程式碼安全：

- ◆ 具備防範螢幕覆蓋攻擊之機制
- ◆ 審視引用之第三方函式庫安全性
- ◆ 輸入欄位須進行型別/長度檢查及防範各類 Injection 攻擊

5

伺服器端 Webview 安全：

- ◆ 連線之網域須通過網頁伺服器弱點掃描 (安全白名單除外)

檢測基準防護目標 II

L2：在 L1 基礎上，強化身分鑑別、授權、Session 管理等防護能力

1

程式發布安全：

- ◆ 提供有效且可實際聯絡之問題回報管道 (如 Email、電話、網頁)

2

敏感性資料保護：

- ◆ 輸入敏感性資料時，須關閉鍵盤快取與自動修正機制
- ◆ 顯示於 UI 之敏感性資料須有遮蔽設計

3

身分鑑別、授權與連線管理安全：

- ◆ 存取個人相關敏感性資料時，須提供身分鑑別與權限授權機制
- ◆ Session ID 須達 128 位元 (含) 以上、無規則性且具備登出失效機制

檢測基準防護目標 III

L3：控管交易期間安全，提升資料保護及畫面側錄之防護能力

1

敏感性資料保護：

- ◆ 執行全程 (包含於關閉/登出後) 均不得將敏感資料以明文存於冗餘檔案或日誌中 (除非以安全加密函式保護)
- ◆ 非使用者主動進行畫面擷取時須主動警示或遮蔽；於 App 切換模式下須主動遮蔽畫面
- ◆ 系統備份檔案中不應存有 App 之敏感性資料

2

交易資源控管安全：

- ◆ 須主動通知交易資訊，並提供拒絕交易機制；交易收款時亦須主動通知
- ◆ 交易前須進行有效之身分鑑別 (如 OTP、生物辨識等)
- ◆ 須提供交易紀錄查詢管道；預授權交易須進行宣告、提供查詢及隨時取消預授權之功能

檢測基準防護目標 IV

F：主動防禦逆向工程、動態偵錯、模擬器環境與系統層破解攻擊

1

防護軟體提示：

- ◆ 於顯著位置 (官網/商店下載頁) 提示使用者安裝行動防護軟體

2

密碼政策強化：

- ◆ 主動提醒密碼至少 8 個字元且避免個人相關資料
- ◆ 主動提醒使用者定期更改密碼 (至多不超過 90 天)

3

系統憑證保護：

- ◆ 將個人可識別資訊、憑證及金鑰儲存於系統憑證設施

4

動態分析及竄改防護：

- ◆ 偵測系統層是否被破解，並警告使用者或終止執行
- ◆ 具備有效之程式碼混淆機制，且無法輕易反混淆還原
- ◆ 偵測執行環境是否為模擬器，及是否開啟 USB 偵錯模式
- ◆ 發布版本必須將 Debug Mode 完全設為關閉

實驗室檢測手法 I

靜態分析：針對 Binary code 進行逆向工程，還原至原始碼加以分析

實驗室查找重點是什麼？常見風險有哪些？

- 檢視 App 設定檔，確認是否開啟不必要之權限，或未關閉 Debug 設定
- 誤將 API Key、Token、加解密金鑰及測試資料等，直接寫死在程式碼中
- 分析引用之第三方函式庫版本，比對是否存有已被公開之已知漏洞 (CVE)

實驗室檢測手法 II

動態分析：透過動態輸入與參數傳遞，檢視 App 的真實行為

實驗室查找重點是什麼？常見風險有哪些？

- 檢查敏感資料於儲存及執行期間是否妥善保護，避免因明文儲存、日誌、快取等殘留造成資料外洩
- 確認是否全程採用安全加密通訊以及實作憑證綁定機制，以降低中間人攻擊、封包竊聽及資料竄改等風險
- 確認是否妥善檢查使用者輸入及外部資料，避免因輸入驗證不足導致注入攻擊、異常操作或未預期錯誤等行為

PART 2

常見開發缺失與防護對策

列舉開發過程最常犯錯的案例與安全做法



不合格案例 I – Hardcoded Secrets

將 Key 或 Token 直接寫死於程式碼或設定檔中

✗ 不合格原因

開發人員為求方便，直接未經任何保護隨即以字串常數形式宣告加解密金鑰或 API Key 等機敏資料

⚡ 面臨衝擊

攻擊者只需使用反編譯工具，即可輕鬆取得加解密金鑰、API Key 等機敏資料

● ApiConfig.java

```
1 public class ApiConfig {
2     // 不安全：金鑰以純文字硬編碼於原始碼中
3     public static final String AWS_SECRET_KEY =
4         "AKIAIOSFODNN7EXAMPLE";
5     public static final String API_SECRET =
6         "sk_live_51Hc9F2eZvKYLo2C0";
7 }
```

※ Android

※ iOS

● APIConfig.swift

```
1 struct APIConfig {
2     // 不安全：常數寫死同樣會被逆向工程取得
3     static let awsSecretKey = "AKIAIOSFODNN7EXAMPLE"
4     static let apiSecret = "sk_live_51Hc9F2eZvKYLo2C0"
5 }
```

不合格案例 I – Hardcoded Secrets

將 Key 或 Token 直接寫死於程式碼或設定檔中

✓ 防護對策

機敏金鑰應由後端動態核發；若須存於裝置本機，須結合 Android Keystore / iOS Keychain 等安全儲存機制

● KeyProvider.java

```
1 // 正確：透過 Android Keystore 產生並保護金鑰，金鑰不落地
2 KeyGenParameterSpec spec = new
  KeyGenParameterSpec.Builder(
3     "secret_key_alias",
4     KeyProperties.PURPOSE_ENCRYPT |
5     KeyProperties.PURPOSE_DECRYPT)
6     .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
7     .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
8     .build();
9 KeyGenerator keyGenerator = KeyGenerator.getInstance(
10    KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");
11 keyGenerator.init(spec);
12 SecretKey secretKey = keyGenerator.generateKey();
```

※ Android

※ iOS

● KeychainStore.swift

```
1 // 正確：使用 Keychain Services，並限制僅本機解鎖時可用
2 let query: [String: Any] = [
3     kSecClass as String: kSecClassGenericPassword,
4     kSecAttrAccount as String: "aws_secret_key",
5     kSecValueData as String: secretData,
6     kSecAttrAccessible as String:
7         kSecAttrAccessibleWhenUnlockedThisDeviceOnly
8 ]
9 SecItemAdd(query as CFDictionary, nil)
```

不合格案例 II – 不安全的資料儲存

敏感性資料未加密保護或使用弱加密演算法

✗ 不合格原因

敏感性資料未經加密保護，直接以明文方式寫入設定檔中，或採用已淘汰之 3DES 等弱加密演算法

⚡ 面臨衝擊

當使用者裝置被破解或遭植入惡意程式時，攻擊者可直接讀取 App 的設定檔或備份檔

● ProfileActivity.java

```
1 // 不安全：明文寫入 SharedPreferences
2 SharedPreferences prefs = getSharedPreferences("user_data",
3 MODE_PRIVATE);
4 prefs.edit()
5   .putString("user_id_card", "A123456789")
6   .apply();
```

※ Android

※ iOS

● ProfileStore.swift

```
1 // 不安全：明文寫入 UserDefaults
2 UserDefaults.standard.set("A123456789", forKey:
3 "user_id_card")
```

不合格案例 II – 不安全的資料儲存

敏感性資料未加密保護或使用弱加密演算法

✅ 防護對策

敏感性資料須加密後才可儲存於本地，並採用 AES-128 以上或 ChaCha20 等加密演算法

SecurePrefs.java

```
1 // 正確：使用 EncryptedSharedPreferences (AES256-GCM)
2 MasterKey masterKey = new MasterKey.Builder(context)
3     .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
4     .build();
5 SharedPreferences prefs =
6     EncryptedSharedPreferences.create(
7         context, "secret_user_prefs", masterKey,
8         EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
9         EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM,
10        prefs.edit().putString("user_id_card", idCard).apply());
```

※ Android

※ iOS

SecureStore.swift

```
1 // 正確：使用 CryptoKit 加密後再存入檔案系統
2 import CryptoKit
3 let key = SymmetricKey(size: .bits256)
4 let sealedBox = try AES.GCM.seal(idCardData, using: key)
5 try sealedBox.combined?.write(to: secureFileURL)
```

不合格案例 III – 不安全的系統日誌

直接將機敏資料輸出至 System Log

✗ 不合格原因

開發階段之除錯訊息於上線前未移除，以致於機敏資料直接透過系統日誌公開

⚡ 面臨衝擊

攻擊者可透過 Logcat 或 Console 工具，直接監聽並攔截發出之機敏資料、User Token 或身分驗證資料等

● AuthService.java

```
1 // 不安全：Release 版本仍印出機敏資訊
2 Log.d("AuthService", "User Token: " +
  user.getAccessToken());
3 Log.d("AuthService", "ID Card: " +
  user.getIdCardNumber());
```

※ Android

※ iOS

● AuthService.swift

```
1 // 不安全：直接使用 print 輸出敏感資訊
2 print("User Token: \(user.accessToken)")
3 print("ID Card: \(user.idCardNumber)")
```

不合格案例 III – 不安全的系統日誌

直接將機敏資料輸出至 System Log

✓ 防護對策

正式版本上線前完整移除除錯用之 Log，或透過規則或系統日誌隱私等機制遮蔽

● proguard-rules.pro

```
1  # 正確：Release 版本自動移除 Log 呼叫
2  -assumenosideeffects class android.util.Log {
3      public static int d(...);
4      public static int v(...);
5  }
```

※Android

※iOS

● AuthService.swift

```
1  // 正確：使用 NSLog，敏感欄位標記為 private
2  import OSLog
3  let logger = Logger(subsystem: "com.mybank.app", category:
    "Auth")
4  logger.debug("User Token: \(user.accessToken, privacy:
    .private)")
5  // Console 實際輸出：User Token: <private>
```

不合格案例 IV – 不安全的網路傳輸

未實作憑證綁定機制或允許明文 HTTP 傳輸

✗ 不合格原因

開發人員自訂信任所有憑證，或於設定檔中允許任意明文連線

⚡ 面臨衝擊

當使用者於公共 Wi-Fi 或受駭之網路環境時，攻擊者可輕易攔截、竊聽或篡改封包資料

● HttpClientConfig.java

```
1 // 不安全：自訂 TrustManager 信任所有憑證
2 TrustManager[] trustAllCerts = new TrustManager[]{
3     new X509TrustManager() {
4         public void checkClientTrusted(X509Certificate[] c,
5             String a) {}
6         public void checkServerTrusted(X509Certificate[] c,
7             String a) {}
8         public X509Certificate[] getAcceptedIssuers() { return
9             null; }
10    }
11 };
```

※ Android

※ iOS

● Info.plist

```
1 <!-- 不安全：允許所有網域使用明文 HTTP -->
2 <key>NSAppTransportSecurity</key>
3 <dict>
4     <key>NSAllowsArbitraryLoads</key>
5     <true/>
6 </dict>
```

不合格案例 IV – 不安全的網路傳輸

未實作憑證綁定機制或允許明文 HTTP 傳輸

✅ 防護對策

採用 TLS 1.2 以上版本，並實作憑證綁定且僅信任預期之伺服器憑證

● NetworkModule.java

```
1 // 正確：使用 OkHttpClient CertificatePinner 綁定憑證公鑰
2 CertificatePinner certificatePinner = new
  CertificatePinner.Builder()
3   .add("api.mybank.com",
4       "sha256/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=")
5   .build();
6 OkHttpClient client = new OkHttpClient.Builder()
7   .certificatePinner(certificatePinner)
8   .build();
```

※ Android

※ iOS

● SessionDelegate.swift

```
1 // 正確：實作 URLSessionDelegate 進行公鑰綁定
2 func urlSession(_ session: URLSession, didReceive
3   challenge: URLAuthenticationChallenge,
4   completionHandler: @escaping
5   (URLSession.AuthChallengeDisposition, URLCredential?) ->
6   Void) {
7   guard let trust =
8     challenge.protectionSpace.serverTrust,
9     isPinnedKeyMatching(trust) else {
10    completionHandler(.cancelAuthenticationChallenge,
11                      nil)
12    return
13  }
14  completionHandler(.useCredential, URLCredential(trust:
15    trust))
16 }
```

不合格案例 V – 不安全的連線管理

未實作 Session 登出失效或逾時機制

✗ 不合格原因

登出機制僅清除本機暫存資料，未通知後端伺服器將對應之 Session 撤銷

⚡ 面臨衝擊

若攻擊者已取得登入 Token，即使使用者已經登出，攻擊者仍可使用該舊 Token 成功呼叫 API 並存取使用者資料

● SessionManager.kt

```
1 // 不安全：登出僅清除本機狀態
2 fun logout() {
3     prefs.edit().clear().apply()
4     startActivity(Intent(this, LoginActivity::class.java))
5     // 後端未同步撤銷，舊 Token 仍可請求 API
6 }
```

※ Android

※ iOS

● SessionManager.swift

```
1 // 不安全：只清除 Keychain，未通知後端撤銷
2 func logout() {
3     KeychainService.remove(forKey: "jwt_token")
4     navigateToLogin()
5 }
```

不合格案例 V – 不安全的連線管理

未實作 Session 登出失效或逾時機制

✅ 防護對策

登出時同步呼叫後端伺服器撤銷授權，確認成功後才清除本機資料

● SessionManager.kt

```
1 // 正確：登出時呼叫 API 撤銷 Token，成功後才清除本機資料
2 fun logout() {
3     apiService.revokeToken(currentToken) { success ->
4         if (success) {
5             prefs.edit().clear().apply()
6             startActivity(Intent(this,
7                 LoginActivity::class.java))
8         }
9     }
10 }
```

※ Android

※ iOS

● SessionManager.swift

```
1 // 正確：確認後端撤銷成功後，才清除本機 Keychain
2 func logout() {
3     apiClient.requestLogout(token: currentToken) { result
4         in
5         switch result {
6         case .success:
7             KeychainService.remove(forKey: "jwt_token")
8             self.navigateToLogin()
9         case .failure(let error):
10             self.showError(error)
11         }
12     }
13 }
```

不合格案例 VI – 螢幕覆蓋攻擊

未主動偵測畫面是否遭其他視窗覆蓋

✗ 不合格原因

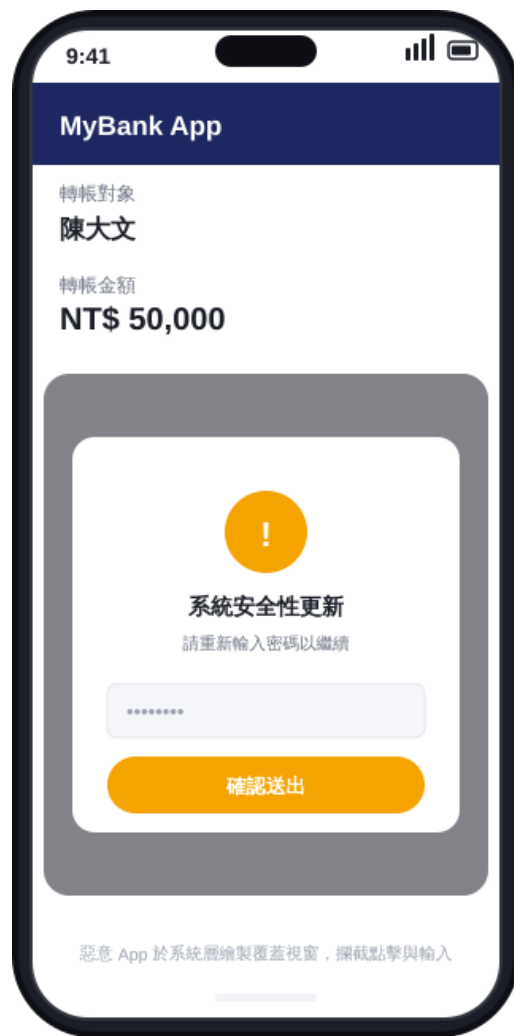
敏感操作在 App 遭覆蓋之狀態下仍可被正常觸發，未主動偵測並提醒使用者

⚡ 面臨衝擊

當惡意 App 取得系統懸浮視窗權限後，可在合法 App 上方覆蓋偽造之對話框，進而誘使使用者操作敏感操作或輸入機敏資料

⚠ 攻擊原理

此類攻擊不依賴程式碼漏洞，而是作業系統層級之視窗顯示機制遭惡意 App 濫用
防禦重點在於「主動偵測畫面異常並彈出警示」，而非單純的資料保護



不合格案例 VI – 螢幕覆蓋攻擊

未主動偵測畫面是否遭其他視窗覆蓋

✅ 防護對策

於觸控事件中檢查覆蓋旗標，當畫面遭覆蓋時忽略所有觸控事件，以確保操作皆在畫面無遮蔽之狀態下完成

● TransferActivity.kt

```
1 // 正確：畫面遭其他視窗覆蓋時，忽略觸控事件（防 Tapjacking）
2 confirmButton.filterTouchesWhenObscured = true
3
4 override fun onTouchEvent(event: MotionEvent): Boolean {
5     if (event.flags and MotionEvent.FLAG_WINDOW_IS_OBSCURED != 0)
6     {
7         return false // 忽略被覆蓋時的觸控事件
8     }
9     return super.onTouchEvent(event)
10 }
```

不合格案例 VII – 不安全的鍵盤快取

自動修正輸入之內容或顯示預測建議字元

✗ 不合格原因

輸入欄位未關閉鍵盤快取與自動修正，以致於機敏資訊被輸入法字典保留

⚡ 面臨衝擊

其他使用者或具備讀取字典權限之第三方輸入法，可輕易推測並取得先前輸入過之機敏資料

● activity_verify.xml

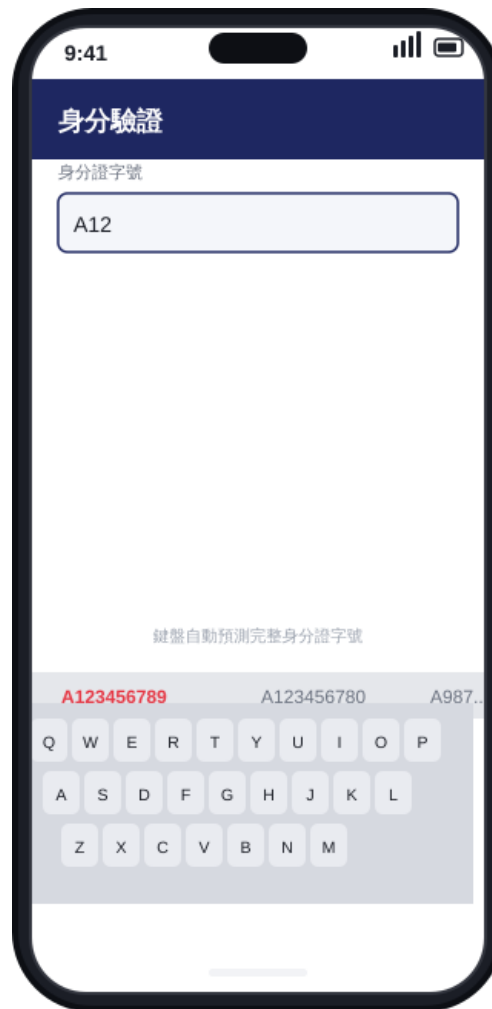
```
1 <!-- 不安全：預設 text 會觸發鍵盤快取與預測 -->
2 <EditText
3     android:id="@+id/idCardInput"
4     android:inputType="text" />
```

※ Android

※ iOS

● VerifyViewController.swift

```
1 // 不安全：預設開啟自動修正功能
2 textField.autocorrectionType = .default
```



不合格案例 VII – 不安全的鍵盤快取

自動修正輸入之內容或顯示預測建議字元

✓ 防護對策

敏感輸入欄位應明確關閉鍵盤快取與自動修正

● activity_verify.xml

```
1 <!-- 正確：關閉鍵盤快取與預測建議 -->
2 <EditText
3     android:id="@+id/idCardInput"
4     android:inputType="textVisiblePassword|textNoSuggestions"
   />
```

※ Android

※ iOS

● VerifyViewController.swift

```
1 // 正確：關閉自動修正，並啟用安全輸入
2 textField.autocorrectionType = .no
3 textField.isSecureTextEntry = true // 自動關閉鍵盤快取
```

不合格案例 VIII – 注入攻擊

未過濾輸入字串導致引發注入攻擊風險

✗ 不合格原因

攻擊語法以字串串接組成，App 未針對輸入做任何驗證，即往後端系統傳送

⚡ 面臨衝擊

攻擊者將惡意指令偽裝成一般資料進行輸入，若 App 未進行輸入驗證或過濾，可能使後端系統誤將其當成合法指令執行，造成資料外洩、資料遭竄改或權限提升等危害

⚠ 攻擊原理

屬於程式碼之輸入驗證缺陷，只要允許任意之外部輸入，攻擊者就能竄改邏輯。防禦重點在於「絕不信任任意輸入」，並一律採用參數化查詢。



不合格案例 VIII – 注入攻擊

未過濾輸入字符串導致引發注入攻擊風險

✓ 防護對策

外部輸入來源可加入白名單驗證，且一律採用參數化語句 (Prepared Statement)

● UserDao.java

```
1 // 錯誤示範 (勿用)：字符串接容易遭 SQL Injection
2 // String sql = "SELECT * FROM users WHERE account='" +
  input + "'";
3
4 // 正確：使用參數化查詢 (Prepared Statement)
5 Cursor cursor = db.rawQuery(
6     "SELECT * FROM users WHERE account = ?",
7     new String[] { inputAccount }
8 );
```

※ Android

※ iOS

● UserDao.swift

```
1 // 正確：使用參數綁定，惡意輸入僅視為純文字比對
2 let query = "SELECT * FROM users WHERE account = ?"
3 sqlite3_prepare_v2(db, query, -1, &statement, nil)
4 sqlite3_bind_text(statement, 1, inputAccount, -1, nil)
```

PART 3

送測前自我評估表 (Checklist)

將檢測基準，濃縮成 Release 前的標準檢驗程序



開發人員自評 Checklist I

開發前與架構設計



確立敏感資料範圍

盤點 App 處理之所有機敏資料（密碼、身分證字號、信用卡號、生物特徵、存取 Token 等），並定義其生命週期。



禁用弱加密演算法

確認全域架構未使用已棄用的演算法（如MD5、SHA1、DES），改用強加密（如AES-256-GCM、SHA-256、RSA-2048、ECC）。



金鑰管理機制

規劃並使用硬體層級之防護（Android Keystore / iOS Keychain）來產生與儲存金鑰。

開發人員自評 Checklist II

資料儲存保護



禁止硬編碼 (Hardcoded)

全域搜尋原始碼，確保未將密碼、API Key 或私鑰以明文寫死在程式碼中。



敏感資料加密儲存

確認寫入 SharedPreferences (Android) 或 UserDefaults (iOS) 等配置檔之機敏資料皆已加密。



資料庫安全

確認 SQLite / CoreData 儲存敏感資料時採用 SQLCipher 等加密機制，且未將檔案存放於非作業系統保護區。

開發人員自評 Checklist III

網路通訊安全



強制加密傳輸

確保連線皆使用 HTTPS，檢查配置檔中是否停用明文 HTTP 傳輸。
(cleartextTrafficPermitted=false/NSAllowsArbitraryLoads=false)



實作憑證綁定 (SSL Pinning)

確保對關鍵之伺服器連線實作憑證綁定，以防止中間人攻擊(MITM)。



憑證驗證機制

檢查網路模組（如 OkHttp、Alamofire），確認未實作信任所有憑證之自訂 TrustManager。

開發人員自評 Checklist IV

身分認證與連線管理



交談識別碼規則

確保交談識別碼長度為 128 位元 (含) 以上，且未與時間、使用者提交資料、具規則性之數字或字串有直接關聯或難以偽造。



安全登出機制

確認使用者點擊登出時，除了清除本機 Token，務必同步呼叫後端 API 註銷該 Token 狀態。



連線逾時處理

確認 App 在處於背景或閒置達一定時間後，會自動登出或要求重新驗證 (如生物辨識等) 。

開發人員自評 Checklist V

介面與互動安全



防範螢幕截圖與錄影

針對顯示機敏資料之頁面，Android 啟用 FLAG_SECURE，iOS 於 App 退至背景時加上遮罩或模糊處理。



關閉鍵盤快取與預測

輸入密碼、身分證等敏感欄位時，設定 textNoSuggestions(Android) 或關閉 autocorrectionType(iOS)。



防範螢幕覆蓋攻擊

確保敏感操作功能頁面（如轉帳、授權）已啟用 filterTouchesWhenObscured = true (Android)，防止惡意覆蓋。

開發人員自評 Checklist VI

函式庫與使用者輸入安全



第三方函式庫盤點與更新

建立軟體物料清單 (SBOM)，定期掃描並更新專案中引用的第三方 SDK 與開源套件，確保無已知的安全漏洞 (CVE)。



嚴格的使用者輸入驗證

所有來自使用者介面、URL Scheme (Deep Link) 等輸入，一律採用「白名單」進行驗證並限制長度，拒絕任何非預期之特殊字元。



防範各類注入攻擊

確保行動應用程式對於輸入之注入攻擊字串，應具備如檢查型別、格式限制等初步之防護設計。

開發人員自評 Checklist VII

系統與送測前最終確認



四大元件安全 (Android)

檢查四大元件 (Activity 、 Service 、 Receiver 、 Provider) ，除非必要，否則 android:exported 應設為 false ，並加上權限控管。



移除 Debug 日誌

確保 Release Build 中所有包含敏感資訊的 Log (如 Log.d, print, NSLog) 皆已被 ProGuard 或編譯設定自動剝離。



關閉除錯模式

確認 AndroidManifest.xml 中的 android:debuggable 設為 false 。

開發人員自評 Checklist VII

系統與送測前最終確認



環境安全檢測

確保 App 具備基本的越獄 (Jailbreak) 與 Root 偵測機制，並在偵測到高風險環境時給予警告或限制功能。



程式碼混淆

確認已正確啟用 ProGuard / R8 或 DexGuard，並檢查混淆後的 Mapping 檔是否完整，防止輕易被逆向工程。

課程地圖



單元三回顧 | 基準說明篇 - 從「理解」到「內化」為開發習慣

深度剖析檢測基準的檢測項目、常見不合格案例，快速掌握送測前自我評估重點

三單元課程地圖

單元一

制度推動篇

打造「全面性防禦」管理力

制度緣起、架構、標章與市場價值

單元二

規範解析篇

建構「開發即合規」防護力

Security by Design · 把規範化為開發控制點

單元三

本單元

基準說明篇

落實「實驗室實戰」檢測實務

檢測項目、常見盲點與送驗前 Checklist



課程主軸 從「外部檢測」→ 到「開發即合規」 以主動防禦貫穿 App 研發全生命週期

十年淬一盾

打造「全面性防禦」管理力

讓每一個 App 標章，都值得被信任

感謝聆聽，敬請指教

行動應用資安聯盟

mas.org.tw

十年淬鍊．全面守護．值得信任

