# COM 402 exercises 2025, session 9: Mobile Security

## Exercise 9.1

You and an app developer agree that native libraries should be used with care in Android apps. As the evening continues, the developer tells you that a couple of months ago, she found a very similar app to one of her apps in the Google Play Store. In fact, the two apps look almost identical. Your blood is pumping faster. This might be a republishing scheme!

Explain to the developer what a republishing scheme is and why it is possible on Android.

## Exercise 9.2

What are the risks of SSL pinning, and how do you mitigate them? You are given the following Kotlin snippet:

```
private fun createCertificatePinner(): CertificatePinner? {
    return if (BuildConfig.DEBUG) {
        // Allow proxy tools like Charles in debug
        null
    } else {
        CertificatePinner.Builder()
            .add("api.myapp.com", PRODUCTION_PIN)
            .build()
    }
}
```

## Exercise 9.3

Your certificate is expiring tomorrow, but you have 100k users on an old app version. How do you handle certificate rotation?"

## Exercise 9.4

You are building an Android app that integrates multiple third-party APIs.

Answer the following:

1. Identify at least three security weaknesses (key storage, reverse engineering, user data exposure).
2. Explain how attackers can extract these secrets using tools like `APKTool` or `JADX`.
3. Propose a secure redesign that removes hardcoded keys and encrypts local data.
4. Discuss why obfuscation is insufficient and when client-side secrets are acceptable.

The following code is currently used in production:

```
object ApiConfig {
    const val MAPS_API_KEY = "AIzaSyXXXXXX-Your-Real-Key"
    const val ANALYTICS_API_KEY = "UA-123456-7"
}

fun getUserData(context: Context): String {
```

```
    // Store user token in SharedPreferences
    val prefs = context.getSharedPreferences("user_prefs", Context.MODE_PRIVATE)
    return prefs.getString("user_token", "") ?: ""
}
```

## Exercise 9.5

App repackaging is a major attack vector: attackers re-sign and redistribute a tampered APK.

1. Describe at least three technical mitigations against repackaging.
2. Consider the following snippet:

   Identify the weakness and propose an improved design.

```
public class MyApp extends Application {
    @Override
    public void onCreate() {
        super.onCreate();

        String expectedSignature =
            "4cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824";
        String currentSig = getMyAppSignature();
        if (!expectedSignature.equals(currentSig)) {
            System.exit(1);
        }
    }
}
```

```
public class MyApp extends Application {
    static {
        System.loadLibrary("appsec");
    }

    private native String getExpectedSignatureHash();

    @Override
    public void onCreate() {
        super.onCreate();

        String expected = getExpectedSignatureHash();
        String current = computeApkSignatureHash();

        if (current == null || !current.equals(expected)) {
            disableApp();
        }
    }
}
```

# Solutions to the Exercises

## Solution 9.1

1. In a republishing scheme, malware authors download an existing app from an app store, reverse engineer it, and add malicious code to it. They publish this stolen app via app stores to follow their nefarious intentions.
2. For example, one way to make money for the malware author is by adding ads to the stolen app.
3. It is possible to reverse engineer Android apps because the code is delivered in an intermediate representation (Dalvik bytecode) that contains most of the original type information. The Dalvik bytecode can, thus, be translated back to a human-readable assembly representation (e.g., smali) or even to the source code (e.g., Java).

## Solution 9.2

1. **Risk: App becomes unusable if pins are wrong.**
   - Test thoroughly using staging servers mirroring production certs.
   - Implement a remote-config kill switch.
2. **Risk: No automatic certificate updates.**
   - Pin intermediate CA certificates instead of leaf certificates.
   - Maintain a planned certificate rotation schedule.
3. **Risk: Debugging becomes difficult.**
   - Use relaxed validation for debug builds.

## Solution 9.3

1. Pin multiple certificates from day one.
2. 3 months before expiry: Deploy app update with both old and new certificate pins
3. 1 month before expiry: Monitor that >95% of users are on the new version
4. Expiry day: Activate new certificate on server
5. 1 month after: Remove old pin in next app update

## Solution 9.4

**Weaknesses:**

- Hard-coded API keys—visible in version control and decompiled APKs.
- User tokens stored in plaintext SharedPreferences.
- No key rotation or backend validation.

Attackers can extract keys by:

- Decompiling the APK with `JADX` to recover Kotlin/Java source.
- Using `APKTool` to read resources and smali code.

**Improved design:**

- Inject secrets at build time via `gradle.properties` or CI environment variables.
- Store long-term secrets server-side; issue short-lived tokens to clients.

- Protect user data using `EncryptedSharedPreferences` and Android Keystore.
- Exclude secret files via `.gitignore`.

**Discussion:**

- Obfuscation (ProGuard/R8) only slows attackers; it does not prevent extraction.
- Client-side storage is acceptable for non-sensitive or short-lived keys.
- CI/CD pipelines with secret managers (Vault, AWS Secrets Manager) prevent key exposure.

## Solution 9.5

**Mitigations against repackaging:**

- Integrity verification (signature hashing, SafetyNet/Play Integrity API).
- Code obfuscation + native libraries for sensitive logic.
- Runtime tamper detection and anti-debugging.
- Build-time watermarking and server-side validation.

**Weakness:** The signature hash is hard-coded; attackers can modify both the check and the reference value.

**Bypass:** Re-sign the app and modify the bytecode to always return true.

**Improved version:**

- Move expected signature to native code or remotely delivered configuration.
- Verify signatures using a hash stored in a native library.
- Integrate the expected hash into the CI pipeline so it is never committed to source.