KonvergeAI

CCT 310 - 3
Application Security

UNIT 5
Android Application Security - II

Application Security Essentials

Android Application Security Best Practices

Enforce secure communication

When you safeguard the data that you exchange between your app and other apps, or between your app and a website, you improve your app's stability and protect the data that you send and receive.

Safeguard communication between apps

To communicate between apps more safely, use implicit intents with an app chooser, signature-based permissions, and non-exported content providers.

```
val intent = Intent(Intent.ACTION_SEND)
val possibleActivitiesList: List<ResolveInfo> =
        packageManager.queryIntentActivities(intent, PackageManager.MATCH_ALL)

// Verify that an activity in at least two apps on the user's device
// can handle the intent. Otherwise, start the intent only if an app
// on the user's device can handle the intent.
if (possibleActivitiesList.size > 1) {

    // Create intent to show chooser.
    // Title is something similar to "Share this photo with."

    val chooser = resources.getString(R.string.chooser_title).let { title ->
        Intent.createChooser(intent, title)
    }
    startActivity(chooser)
} else if (intent.resolveActivity(packageManager) != null) {
    startActivity(intent)
}
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
    <permission android:name="my_custom_permission_name"
                android:protectionLevel="signature" />
```

Disallow access to your app's content providers

Unless you intend to send data from your app to a different app that you don't own, explicitly disallow other developers' apps from accessing your app's ContentProvider objects. This setting is particularly important if your app can be installed on devices running Android 4.1.1 (API level 16) or lower, as the android:exported attribute of the <provider> element is true by default on those versions of Android.

Ask for credentials before showing sensitive information

When requesting credentials from users so that they can access sensitive information or premium content in your app, ask for either a PIN/password/pattern or a biometric credential, such as face recognition or fingerprint recognition.

Apply network security measures / Best Practices to enforce network security measures

Use TLS traffic

If your app communicates with a web server that has a certificate issued by a well-known, trusted certificate authority (CA)

Add a network security configuration

If your app uses new or custom CAs, you can declare your network's security settings in a configuration file. This process lets you create the configuration without modifying any app code.

Use WebView objects carefully

WebView objects in your app shouldn't let users navigate to sites that are outside of your control. Whenever possible, use an allowlist to restrict the content loaded by your app's WebView objects.

In addition, never enable JavaScript interface support unless you completely control and trust the content in your app's WebView objects.

Provide the right permissions

Request only the minimum number of permissions necessary for your app to function properly. When possible, relinquish permissions when your app no longer needs them.

Use intents to defer permissions

Whenever possible, don't add a permission to your app to complete an action that can be completed in another app. Instead, use an intent to defer the request to a different app that already has the necessary permission.

Share data securely across apps

Follow these best practices to share your app's content with other apps in a more secure manner:

•Enforce read-only or write-only permissions as needed.

•Provide clients one-time access to data by using the FLAG_GRANT_READ_URI_PERMISSION and FLAG_GRANT_WRITE_URI_PERMISSION flags.

•When sharing data, use content:// URIs, not file:// URIs. Instances of FileProvider do this for you.

Store data safely

• Although your app might require access to sensitive user information, users grant your app access to their data only if they trust that you safeguard it properly.

• Store private data within internal storage

• Store all private user data within the device's internal storage, which is sandboxed per app. Your app doesn't need to request permission to view these files, and other apps can't access the files. As an added security measure, when the user uninstalls an app, the device deletes all files that the app saved within internal storage.

Store data in external storage based on use case

Use external storage for large, non-sensitive files that are specific to your app as well as files that your app shares with other apps. The specific APIs that you use depend on whether your app is designed to access app-specific files or access shared files.

If a file doesn't contain private or sensitive information but provides value to the user only in your app, store the file in an app-specific directory on external storage.

If your app needs to access or store a file that provides value to other apps, use one of the following APIs, depending on your use case:

•**Media files:** To store and access images, audio files, and videos that are shared between apps, use the Media Store API.

•**Other files:** To store and access other types of shared files, including downloaded files, use the Storage Access Framework.

Check availability of storage volume

•If your app interacts with a removable external storage device, keep in mind that the user might remove the storage device while your app is trying to access it. Include logic to verify that the storage device is available.

Check validity of data

•If your app uses data from external storage, make sure that the contents of the data haven't been corrupted or modified. Include logic to handle files that are no longer in a stable format.

•Keep Third party dependencies updated

Permissions & Policy File

Add declaration to app manifest

To declare a permission that your app might request, include the appropriate `<uses-permission>` element in your app's manifest file. For example, an app that needs to access the camera has this line in `AndroidManifest.xml`:

```
<manifest ...>
    <uses-permission android:name="android.permission.CAMERA"/>
    <application ...>
        ...
    </application>
</manifest>
```

Increased situational context

Users are prompted at runtime, in the context of your app, for permission to access the functionality covered by those permission groups. Users are more sensitive to the context in which the permission is requested, and if there's a mismatch between what you are requesting and the purpose of your app, it's even more important to provide detailed explanation to the user as to why you're requesting the permission. Whenever possible, you should provide an explanation of your request both at the time of the request and in a follow-up dialog if the user denies the request.

Greater flexibility in granting permissions

Users can deny access to individual permissions at the time they're requested *and* in settings, but they may still be surprised when functionality is broken as a result. It's a good idea to monitor how many users are denying permissions (e.g. using Google Analytics) so that you can either refactor your app to avoid depending on that permission or provide a better explanation of why you need the permission for your app to work properly. You should also make sure that your app handles exceptions when users deny permission requests or toggle off permissions in settings.

Increased transactional burden

Users are asked to grant access for permission groups individually and not as a set. This makes it extremely important to minimize the number of permissions you're requesting. This increases the user-burden for granting permissions and therefore increases the probability that at least one of the requests will be denied.

Limit background access to location

When your app is running in the background, access to location should be critical to the app's core functionality and show a clear benefit to users.

Crypto APIs

Android cryptography APIs are based on the Java Cryptography Architecture (JCA). JCA separates the interfaces and implementation, making it possible to include several security providers that can implement sets of cryptographic algorithms. Most of the JCA interfaces and classes are defined in the java.security.* and javax.crypto.* packages. In addition, there are Android specific packages android.security.* and android.security.keystore.*.

KeyStore and KeyChain provide APIs for storing and using keys (behind the scene, KeyChain API uses KeyStore system). These systems allow to administer the full lifecycle of the cryptographic keys.KeyStore and KeyChain provide APIs for storing and using keys (behind the scene, KeyChain API uses KeyStore system). These systems allow to administer the full lifecycle of the cryptographic keys.

- generating a key
- using a key
- storing a key
- archiving a key
- deleting a key

Usages :-

1. Any Encryption/Decryption
2. Random Number Generation
3. for signing/verifying - to ensure integrity of data (as well as accountability in some cases)

Securing Application Data

Techniques to explain access to more sensitive information

The permissions related to location, microphone, and camera grant your app access to particularly sensitive information about users. The platform includes several mechanisms, to help users stay informed and in control over which apps can access location, microphone, and camera.
These privacy-preserving system features shouldn't affect how your app handles the permissions related to location, microphone, and camera, as long as you follow privacy best practices.

In particular, make sure you do the following techniques in your app:
•Wait to access the device's camera until the user has granted the CAMERA permission to your app.
•Wait to access the device's microphone until the user has granted the RECORD_AUDIO permission to your app.
•Wait until the user interacts with a feature in your app that requires location before you request the ACCESS_COARSE_LOCATION permission or the ACCESS_FINE_LOCATION permission, as described in the guide on how to request location permissions.
•Wait until the user grants your app either the ACCESS_COARSE_LOCATION permission or the ACCESS_FINE_LOCATION permission before you request the ACCESS_BACKGROUND_LOCATION permission.

Techniques to Store Data Securely in Android Applications

The most common security concern for an application on Android is whether the data that you save on the device is accessible to other apps. There are three fundamental ways to save data on the device:

- Internal storage
- External storage
- Content providers

The following sections describe the security issues associated with each approach.

Internal storage

By default, files that you create on internal storage are accessible only to your app. Android implements this protection, and it's sufficient for most applications.

Avoid the deprecated MODE_WORLD_WRITEABLE and MODE_WORLD_READABLE modes for IPC files. They don't provide the ability to limit data access to particular applications, and they don't provide any control of data format. If you want to share your data with other app processes, consider using a content provider instead, which offers read and write permissions to other apps and can make dynamic permission grants on a case-by-case basis.

To provide additional protection for sensitive data, you can encrypt local files using the Security library. This measure can provide protection for a lost device without file system encryption.

Techniques to Store Data Securely in Android Applications

External storage

Files created on external storage, such as SD cards, are globally readable and writable. Because external storage can be removed by the user and also modified by any application, only store non-sensitive information using external storage.

To read and write files on external storage in a more secure way, consider using the Security library, which provides the EncryptedFile class.

Perform input validation when handling data from external storage as you would with data from any untrusted source. Don't store executables or class files on external storage prior to dynamic loading. If your app does retrieve executable files from external storage, make sure the files are signed and cryptographically verified prior to dynamic loading.

Techniques to Store Data Securely in Android Applications

Content providers offer a structured storage mechanism that can be limited to your own application or exported to allow access by other applications.

If you don't intend to provide other applications with access to your ContentProvider, mark it as android:exported=false in the application manifest. Otherwise, set the android:exported attribute to true to let other apps access the stored data.

When creating a ContentProvider that is exported for use by other applications, you can specify a single permission for reading and writing, or you can specify distinct permissions for reading and writing. Limit your permissions to those required to accomplish the task at hand. Keep in mind that it’s usually easier to add permissions later to expose new functionality than it is to take them away and impact existing users.

If you are using a content provider for sharing data between only your own apps, we recommend using the android:protectionLevel attribute set to signature protection. Signature permissions don't require user confirmation, so they provide a better user experience and more controlled access to the content provider data when the apps accessing the data are signed with the same key.

Securing Server Interactions

A server with a TLS certificate has a public key and a matching private key. The server uses public-key cryptography to sign its certificate during the TLS handshake.

A simple handshake only proves that the server knows the certificate's private key. To address this situation, let the client trust multiple certificates. A given server is untrustworthy if its certificate doesn't appear in the client-side set of trusted certificates.

However, servers might use key rotation to change their certificate's public key with a new one. The server configuration change necessitates updating the client app. If the server is a third-party web service, such as a web browser or email app, it's more difficult to know when to update the client app.

Servers usually rely on Certificate Authorities (CAs) certificates to issue certificates, which keeps the client-side configuration more stable over time. A CA signs a server certificate using its private key. The client can then check that the server has a platform-known CA certificate.

Trusted CAs are usually listed on the host platform. Android 8.0 (API level 26) includes over 100 CAs that are updated in each version and don't change between devices.

Client apps need a mechanism to verify the server because the CA offers certificates for numerous servers. The CA's certificate identifies the server using either a specific name, such as *gmail.com*, or using a wildcard, such as **.google.com*.

Future of Android App Security

Two-Factor Authentication will become the de facto standard
Phone logins will start to employ multi-factor authentication
More frequent (and smaller) security updates
Sideloading applications no longer possible
Application vetting will become considerably more rigorous

Case Study

Megabank wants to build an android city transportation application for its consumers. Construct a list of Best Practices to be followed for security and privacy which can be used by developers of the application