Improve your app's security

By making your app more secure, you help preserve user trust and device integrity.

This page presents several best practices that have a significant, positive impact on your app's security.

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.

Show an app chooser

If an implicit intent can launch at least two possible apps on a user's device, explicitly show an app chooser. This interaction strategy lets users transfer sensitive information to an app that they trust.

KotlinJava (#java)
(#kotlin)

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

// 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)
    }
```

Related info:

- Show an app chooser (/training/basics/intents/sending#AppChooser)
- `Intent` (/reference/android/content/Intent)

Apply signature-based permissions

When sharing data between two apps that you control or own, use *signature-based* permissions. These permissions don't require user confirmation and instead check that the apps accessing the data are signed using the same signing key. Therefore, these permissions offer a more streamlined, secure user experience.

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

Related info:

- Sign your app (/studio/publish/app-signing)
- `android:protectionLevel` (/guide/topics/manifest/permission-element#plevel)

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` (/reference/android/content/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` (/guide/topics/manifest/provider-element#exported) attribute of the `<provider>` (/guide/topics/manifest/provider-element) element is `true` by default on those versions of Android.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
    <application ... >
        <provider
            android:name="android.support.v4.content.FileProvider"
            android:authorities="com.example.myapp.fileprovider"
            ...
            android:exported="false">
            <!-- Place child elements of <provider> here. -->
        </provider>
        ...
    </application>
</manifest>
```

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.

To learn more about how to request biometric credentials, see the guide about biometric authentication (/training/sign-in/biometric-auth).

Apply network security measures

The following sections describe how you can improve your app's network security.

Use TLS traffic

If your app communicates with a web server that has a certificate issued by a well-known, trusted certificate authority (CA), use an HTTPS request like the following:

KotlinJava (#java)
(#kotlin)

```
val url = URL("https://www.google.com")
val urlConnection = url.openConnection() as HttpsURLConnection
urlConnection.connect()
urlConnection.inputStream.use {
    ...
}
```

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.

To add a network security configuration file to your app, follow these steps:

1. Declare the configuration in your app's manifest:

```
<manifest ... >
    <application
        android:networkSecurityConfig="@xml/network_security_config"
        ... >
        <!-- Place child elements of <application> element here. -->
    </application>
</manifest>
```

2. Add an XML resource file, located at `res/xml/network_security_config.xml`.

   Specify that all traffic to particular domains must use HTTPS by disabling clear-text:

```
<network-security-config>
    <domain-config cleartextTrafficPermitted="false">
        <domain includeSubdomains="true">secure.example.com</domain>
        ...
    </domain-config>
</network-security-config>
```

During the development process, you can use the `<debug-overrides>` element to explicitly allow user-installed certificates. This element overrides your app's security-critical options during debugging and testing without affecting the app's release configuration. The following snippet shows how to define this element in your app's network security configuration XML file:

```
<network-security-config>
    <debug-overrides>
        <trust-anchors>
            <certificates src="user" />
        </trust-anchors>
    </debug-overrides>
</network-security-config>
```

Related info: Network security configuration (/training/articles/security-config)

Create your own trust manager

Your TLS checker shouldn't accept every certificate. You might need to set up a trust manager and handle all TLS warnings that occur if one of the following conditions applies to your use case:

- You're communicating with a web server that has a certificate signed by a new or custom CA.
- That CA isn't trusted by the device you're using.
- You can't use a network security configuration (#network-security-config).

To learn more about how to complete these steps, see the discussion about handling an unknown certificate authority (/training/articles/security-ssl#UnknownCa).

Related info:

- Security with network protocols (/training/articles/security-ssl)
- `CertificateFactory` (/reference/java/security/cert/CertificateFactory)
- `HttpsURLConnection` (/reference/javax/net/ssl/HttpsURLConnection)
- `TrustManager` (/reference/javax/net/ssl/TrustManager)

Use WebView objects carefully

`WebView` (/reference/android/webkit/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 (/guide/webapps/webview#UsingJavaScript) unless you completely control and trust the content in your app's `WebView` objects.

Use HTML message channels

If your app must use JavaScript interface support on devices running Android 6.0 (API level 23) and higher, use HTML message channels instead of communicating between a website and your app, as shown in the following code snippet:

Kotlin Java (#java)
(#kotlin)

```
val myWebView: WebView = findViewById(R.id.webview)

// channel[0] and channel[1] represent the two ports.
// They are already entangled with each other and have been started.
val channel: Array<out WebMessagePort> = myWebView.createWebMessageChannel

// Create handler for channel[0] to receive messages.
channel[0].setWebMessageCallback(object : WebMessagePort.WebMessageCallbacl
```

```
    override fun onMessage(port: WebMessagePort, message: WebMessage) {
        Log.d(TAG, "On port $port, received this message: $message")
    }
})

// Send a message from channel[1] to channel[0].
channel[1].postMessage(WebMessage("My secure message"))
```

Related info:

- `WebMessage` (/reference/android/webkit/WebMessage)
- `WebMessagePort` (/reference/android/webkit/WebMessagePort)

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.

The following example shows how to use an intent to direct users to a contacts app instead of requesting the `READ_CONTACTS` (/reference/android/Manifest.permission#READ_CONTACTS) and `WRITE_CONTACTS` (/reference/android/Manifest.permission#WRITE_CONTACTS) permissions:

KotlinJava (#java)
(#kotlin)

```
// Delegates the responsibility of creating the contact to a contacts app,
// which has already been granted the appropriate WRITE_CONTACTS permissio
Intent(Intent.ACTION_INSERT).apply {
    type = ContactsContract.Contacts.CONTENT_TYPE
}.also { intent ->
    // Make sure that the user has a contacts app installed on their devic
```

```
    intent.resolveActivity(packageManager)?.run {
        startActivity(intent)
    }
}
```

In addition, if your app needs to perform file-based I/O—such as accessing storage or choosing a file—it doesn't need special permissions because the system can complete the operations on your app's behalf. Better still, after a user selects content at a particular URI, the calling app gets granted permission to the selected resource.

Related info:

- Common Intents (/guide/components/intents-common)
- `Intent` (/reference/android/content/Intent)

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` (/reference/android/content/Intent#FLAG_GRANT_READ_URI_PERMISSION) and `FLAG_GRANT_WRITE_URI_PERMISSION` (/reference/android/content/Intent#FLAG_GRANT_WRITE_URI_PERMISSION) flags.
- When sharing data, use `content://` URIs, not `file://` URIs. Instances of `FileProvider` (/reference/androidx/core/content/FileProvider) do this for you.

The following code snippet shows how to use URI permission grant flags and content provider permissions to display an app's PDF file in a separate PDF viewer app:

Kotlin Java (#java)
(#kotlin)

```
// Create an Intent to launch a PDF viewer for a file owned by this app.
Intent(Intent.ACTION_VIEW).apply {
```

```
    data = Uri.parse("content://com.example/personal-info.pdf")

    // This flag gives the started app read access to the file.
    addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
}.also { intent ->
    // Make sure that the user has a PDF viewer app installed on their dev
    intent.resolveActivity(packageManager)?.run {
        startActivity(intent)
    }
}
```

Note: Executing files from the writable app home directory is a W^X violation (https://en.wikipedia.org/wiki/W%5EX). For this reason, untrusted apps that target Android 10 (API level 29) and higher can't invoke `exec()` on files within the app's home directory, only the binary code that's embedded within an app's APK file. In addition, apps that target Android 10 and higher can't, in memory, modify executable code from files that have been opened with `dlopen()`. This includes any shared object (`.so`) files with text relocations.

Related info: `android:grantUriPermissions` (/guide/topics/manifest/provider-element#gprmsn)

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.

The following code snippet demonstrates one way to write data to internal storage:

Kotlin Java (#java)

```
// Creates a file with this name, or replaces an existing file
// that has the same name. Note that the file name cannot contain
// path separators.
val FILE_NAME = "sensitive_info.txt"
val fileContents = "This is some top-secret information!"
File(filesDir, FILE_NAME).bufferedWriter().use { writer ->
    writer.write(fileContents)
}
```

The following code snippet shows the inverse operation, reading data from internal storage:

Kotlin Java (#java)
(#kotlin)

```
val FILE_NAME = "sensitive_info.txt"
val contents = File(filesDir, FILE_NAME).bufferedReader().useLines { lines
    lines.fold("") { working, line ->
        "$working\n$line"
    }
}
```

Related info:

- Data and file storage overview (/guide/topics/data/data-storage)
- `FileInputStream` (/reference/java/io/FileInputStream)
- `FileOutputStream` (/reference/java/io/FileOutputStream)
- `Context.MODE_PRIVATE` (/reference/android/content/Context#MODE_PRIVATE)

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 (/training/data-storage/app-specific#external).

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 (/training/data-storage/shared/media).
- **Other files:** To store and access other types of shared files, including downloaded files, use the Storage Access Framework (/training/data-storage/shared/documents-files).

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 (/training/data-storage/app-specific#external-verify-availability).

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.

The following code snippet includes an example of a hash verifier:

Kotlin Java (#java)
(#kotlin)

```
val hash = calculateHash(stream)
// Store "expectedHash" in a secure location.
if (hash == expectedHash) {
    // Work with the content.
}

// Calculating the hash code can take quite a bit of time, so it shouldn't
// be done on the main thread.
suspend fun calculateHash(stream: InputStream): String {
    return withContext(Dispatchers.IO) {
```

```
        val digest = MessageDigest.getInstance("SHA-512")
        val digestStream = DigestInputStream(stream, digest)
        while (digestStream.read() != -1) {
            // The DigestInputStream does the work; nothing for us to do.
        }
        digest.digest().joinToString(":") { "%02x".format(it) }
    }
}
```

Store only non-sensitive data in cache files

To provide faster access to non-sensitive app data, store it in the device's cache. For caches larger than 1 MB, use `getExternalCacheDir()` (/reference/android/content/Context#getExternalCacheDir()). For caches 1 MB or smaller, use `getCacheDir()` (/reference/android/content/Context#getCacheDir()). Both methods provide you with the `File` (/reference/java/io/File) object that contains your app's cached data.

The following code snippet shows how to cache a file that your app recently downloaded:

Kotlin Java (#java)
(#kotlin)

```
val cacheFile = File(myDownloadedFileUri).let { fileToCache ->
    File(cacheDir.path, fileToCache.name)
}
```

Note: If you use `getExternalCacheDir()` (/reference/android/content/Context#getExternalCacheDir()) to place your app's cache within shared storage, the user might eject the media containing this storage while your app is running. Include logic to gracefully handle the cache miss that this user behavior causes.

Caution: There is no security enforced on these files. Therefore, any app that targets Android 10 (API level 29) or lower and has the `WRITE_EXTERNAL_STORAGE` (/reference/android/Manifest.permission#WRITE_EXTERNAL_STORAGE) permission can access the contents of this cache.

Related info: Data and file storage overview (/guide/topics/data/data-storage)

Use SharedPreferences in private mode

When using `getSharedPreferences()` (/reference/android/content/Context#getSharedPreferences(java.lang.String, int)) to create or access your app's `SharedPreferences` (/reference/android/content/SharedPreferences) objects, use `MODE_PRIVATE` (/reference/android/content/Context#MODE_PRIVATE). That way, only your app can access the information within the shared preferences file.

If you want to share data across apps, don't use `SharedPreferences` objects. Instead, follow the steps to share data securely across apps (#permissions-share-data).

Related info:

- Data and file storage overview (/guide/topics/data/data-storage)

Keep services and dependencies up to date

Most apps use external libraries and device system information to complete specialized tasks. By keeping your app's dependencies up to date, you make these points of communication more secure.

Check the Google Play services security provider

Note: This section applies only to apps targeting devices that have Google Play services (https://developers.google.com/android/guides/overview) installed.

If your app uses Google Play services, make sure that it's updated on the device where your app is installed. Perform the check asynchronously, off of the UI thread. If the device isn't up to date, trigger an authorization error.

To determine whether Google Play services is up to date on the device where your app is installed, follow the steps in the guide about Updating your security provider to protect against SSL exploits (/training/articles/security-gms-provider).

Related info:

- `ProviderInstaller` (https://developers.google.com/android/reference/com/google/android/gms/security/ProviderInstaller)
- `ProviderInstaller.ProviderInstallListener` (https://developers.google.com/android/reference/com/google/android/gms/security/ProviderInstaller.ProviderInstallListener)

Update all app dependencies

Before deploying your app, make sure that all libraries, SDKs, and other dependencies are up to date:

- For first-party dependencies, such as the Android SDK, use the updating tools found in Android Studio, such as the SDK Manager (/studio/intro/update#sdk-manager).
- For third-party dependencies, check the websites of the libraries that your app uses, and install any available updates and security patches.

Related info: Add build dependencies (/studio/build/dependencies#google_and_android_support_repositories)

More information

To learn more about how to make your app more secure, view the following resources:

- Core app quality security checklist (/distribute/essentials/quality/core#sc)
- App security improvement program (/google/play/asi)
- Android Developers channel on YouTube (https://www.youtube.com/user/androiddevelopers)
- Android Protected Confirmation: Taking transaction security to the next level (https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html)

Last updated 2025-11-21 UTC.