App permissions best practices

Permission requests protect sensitive information available from a device and should only be used when access to information is necessary for the functioning of your app. This document provides tips on ways you might be able to achieve the same (or better) functionality without requiring access to such information; it is not an exhaustive discussion of how permissions work in the Android operating system.

For a more general look at Android permissions, please see Permissions overview (https://developer.android.com/guide/topics/permissions/requesting.html). For details on how to work with permissions in your code, see Requesting app permissions (https://developer.android.com/training/permissions/requesting.html).

Tenets of working with Android permissions

We recommend following these tenets when working with Android permissions:

#1: Only use the permissions necessary for your app to work. Depending on how you are using the permissions, there may be another way to do what you need (system intents, identifiers, backgrounding for phone calls) without relying on access to sensitive information.

#2: Pay attention to permissions required by libraries. When you include a library, you also inherit its permission requirements. You should be aware of what you're including, the permissions they require, and what those permissions are used for.

#3: Be transparent. When you make a permissions request, be clear about what you're accessing, and why, so users can make informed decisions. Make this information available alongside the permission request including install, runtime, or update permission dialogues.

#4: Make system accesses explicit. Providing continuous indications when you access sensitive capabilities (for example, the camera or microphone) makes it clear to users when you're collecting data and avoids the perception that you're collecting data surreptitiously.

The remaining sections of this guide elaborate on these rules in the context of developing Android applications.

Permissions in Android 6.0+

Android 6.0 Marshmallow introduced a new permissions model that lets apps request permissions from the user at runtime, rather than prior to installation. Apps that support the new model request permissions when the app actually requires the services or data protected by the services. While this doesn't (necessarily) change overall app behavior, it does create a few changes relevant to the way sensitive user data is handled:

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 created when users deny permission requests or toggle off permissions in settings.

Increased transactional burden: Users will be 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 because it increases the user burden for granting permissions and increases the probability that at least one of the requests will be denied.

Avoid requesting unnecessary permissions

Every time you ask for a permission, you force the user to make a decision. You should minimize the number of times you make these requests. If the user is running Android 6.0 (API level 23) or later, every time the user tries some new app feature that requires a permission, the app has to interrupt the user's work with a permission request. If the user is running an earlier version of Android, the user has to grant every one of the app's permissions when installing the app; if the list is too long or seems inappropriate, the user may decide not to install your app at all. For these reasons, you should minimize the number of permissions your app needs.

This section provides alternatives to common use-cases that will help you limit the number of permission requests you make. Since the number and type of user-surfaced permissions

requested affects downloads compared to other similar apps requesting fewer permissions, it's best to avoid requesting permissions for unnecessary functionality.

Use an intent instead

In many cases, you can choose between two ways for your app to perform a task. Your app ask for permission to perform the task itself or it can use an *intent* to have another app perform the task.

For example, suppose your app needs to be able to take pictures with the device camera. Your app can request the `CAMERA` (https://developer.android.com/reference/android/Manifest.permission.html#CAMERA) permission, which allows your app to access the camera directly. Your app would then use the camera APIs to control the camera and take a picture. This approach gives your app full control over the photography process, and lets you incorporate the camera UI into your app.

However, if your requirement for access to user data is infrequent — in other words, it's not unacceptably disruptive for the user to be presented with a runtime dialogue each time you need to access data — you can use an *intent based request*. Android provides some system intents that applications can use without requiring permissions because the user chooses what, if anything, to share with the app at the time the intent based request is issued.

For example, an intent action type of `MediaStore.ACTION_IMAGE_CAPTURE` (https://developer.android.com/reference/android/provider/MediaStore.html#ACTION_IMAGE_CAPTURE)
or `MediaStore.ACTION_VIDEO_CAPTURE` (https://developer.android.com/reference/android/provider/MediaStore.html#ACTION_VIDEO_CAPTURE)
can be used to capture images or videos without directly using the Camera (https://developer.android.com/reference/android/hardware/Camera.html) object (or requiring the permission). In this case, the system intent will ask for the user's permission on your behalf every time an image is captured.

Similarly, if you need to make a phone call, access the user's contacts, and so on, you can do that by creating an appropriate intent, or you can request the permission and access the appropriate objects directly. There are advantages and disadvantages to each approach.

If you use permissions:

- Your app has full control over the user experience when you perform the operation. However, such broad control adds to the complexity of your code, since you need to design an appropriate UI.

- The user is prompted to give permission once, either at run time or at install time (depending on the user's Android version). After that, your app can perform the operation without requiring additional interaction from the user. However, if the user doesn't grant the permission (or revokes it later on), your app loses the ability to perform the operation at all.

If you use an intent:

- You do not have to design the UI for the operation. The app that handles the intent provides the UI.
- The user can use their preferred app for the task. For example, the user can select their favorite photo app to take a picture.
- If the user does not have a default app for the operation, the system prompts the user to choose an app. If the user does not designate a default handler, they may have to go through an extra dialog every time they perform the operation.

Don't overwhelm the user

If the user is running Android 6.0 (API level 23) or later, the user has to grant your app its permissions while they are running the app. If you confront the user with a lot of requests for permissions at once, you may overwhelm the user and cause them to quit your app. Instead, you should ask for permissions as you need them.

In some cases, one or more permissions might be absolutely essential to your app. It might make sense to ask for all of those permissions as soon as the app launches. For example, if you make a photography app, the app would need access to the device camera. When the user launches the app for the first time, they won't be surprised to be asked for permission to use the camera. But if the same app also had a feature to share photos with the user's contacts, you probably should *not* ask for the `READ_CONTACTS` (https://developer.android.com/reference/android/Manifest.permission.html#READ_CONTACTS) permission at first launch. Instead, wait until the user tries to use the "sharing" feature and ask for the permission then.

If your app provides a tutorial, it may make sense to request the app's essential permissions at the end of the tutorial sequence.

Pause media after losing audio focus

In this case, your application needs to go into the background when the user gets a phone call and refocus only once the call stops.

The common approach in these cases - for example, a media player muting or pausing during a phone call - is to listen for changes in the call state using `PhoneStateListener` or listening for the broadcast of `android.intent.action.PHONE_STATE`. The problem with this solution is that it requires the `READ_PHONE_STATE` permission, which forces the user to grant access to a wide cross section of sensitive data such as their device and SIM hardware IDs and the phone number of the incoming call.

You can avoid this by requesting `AudioFocus` for your app, which doesn't require explicit permissions (because it does not access sensitive information). Simply put the code required to background your audio in the `onAudioFocusChange()` (https://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html#onAudioFocusChange(int)) event handler and it will run automatically when the OS shifts its audio focus. More detailed documentation on how to do this can be found here (https://developer.android.com/training/managing-audio/audio-focus.html).

Determine the device your instance is running on

In this case, you need a unique identifier to determine which device the instance of your app is running on.

Applications may have device-specific preferences or messaging (e.g., saving a device-specific playlist for a user in the cloud so that they can have a different playlist for their car and at home). A common solution is to leverage device identifiers such as `Device IMEI`, but this requires the `Device ID and call information` permission group (`PHONE` in M+). It also uses an identifier which cannot be reset and is shared across all apps.

There are two alternatives to using these types of identifiers:

1. Use the `com.google.android.gms.iid` InstanceID API. `getInstance(Context context).getID()` will return a unique device identifier for your application instance. The result is an app instance scoped identifier that can be used as a key when storing information about the app and is reset if the user re-installs the app.
2. Create your own identifier that's scoped to your app's storage using basic system functions like `randomUUID()` (https://developer.android.com/reference/java/util/UUID.html#randomUUID()).

Create a unique identifier for advertising or user analytics

In this case, you need a unique identifier for building a profile for users who are not signed in to your app (e.g., for ads targeting or measuring conversions).

Building a profile for advertising and user analytics sometimes requires an identifier that is shared across other applications. Common solutions for this involve leveraging device identifiers such as `Device IMEI`, which requires the `Device ID and call information` permission group (`PHONE` in API level 23+) and cannot be reset by the user. In any of these cases, in addition to using a non-resettable identifier and requesting a permission that might seem unusual to users, you will also be in violation of the Play Developer Program Policies (https://play.google.com/about/developer-content-policy.html).

Unfortunately, in these cases using the `com.google.android.gms.iid` InstanceID API or system functions to create an app-scoped ID are not appropriate solutions because the ID may need to be shared across apps. An alternative solution is to use the `Advertising Identifier` available from the `AdvertisingIdClient.Info` (https://developer.android.com/reference/com/google/android/gms/ads/identifier/AdvertisingIdClient.Info.html) class via the `getId()` method. You can create an `AdvertisingIdClient.Info` object using the `getAdvertisingIdInfo(Context)` method and call the `getId()` method to use the identifier. ***Note that this method is blocking***, so you should not call it from the main thread; a detailed explanation of this method is available here (https://developer.android.com/google/play-services/id.html).

Know the libraries you're working with

Sometimes permissions are required by the libraries you use in your app. For example, ads and analytics libraries may require access to the `Location` or `Identity` permissions groups to implement the required functionality. But from the user's point of view, the permission request comes from your app, not the library.

Just as users select apps that use fewer permissions for the same functionality, developers should review their libraries and select third-party SDKs that are not using unnecessary permissions. For example, try to avoid libraries that require the `Identity` permission group unless there is a clear user-facing reason why the app needs those permissions. In particular, for libraries that provide location functionality, make sure you are not required to request the `FINE_LOCATION` permission unless you are using location-based targeting functionality.

Explain why you need permissions

The permissions dialog shown by the system when you call **`requestPermissions()`** (https://developer.android.com/reference/android/support/v4/app/ActivityCompat.html#requestPermissions(android.app.Activity, java.lang.String[], int)) says what permission your app wants, but doesn't say why. In some cases, the user may find that puzzling. It's a good idea to explain to the user why your app wants the permissions before calling **`requestPermissions()`** (https://developer.android.com/reference/android/support/v4/app/ActivityCompat.html#requestPermissions(android.app.Activity, java.lang.String[], int)).

Research shows that users are much more comfortable with permissions requests if they know why the app needs them. A user study showed that:

> *...a user's willingness to grant a given permission to a given mobile app is strongly influenced by the purpose associated with such a permission. For instance a user's willingness to grant access to his or her location will vary based on whether the request is required to support the app's core functionality or whether it is to share this information with an advertising network or an analytics company.*[1] (#references)

Based on his group's research, Professor Jason Hong from CMU concluded that, in general:

> *...when people know why an app is using something as sensitive as their location — for example, for targeted advertising — it makes them more comfortable than when simply told an app is using their location.*[1] (#references)

As a result, if you're only using a fraction of the API calls that fall under a permission group, it helps to explicitly list which of those permissions you're using, and why. For example:

- If you're only using coarse location, let the user know this in your app description or in help articles about your app.
- If you need access to SMS messages to receive authentication codes that protect the user from fraud, let the user know this in your app description and/or the first time you access the data.

★ **Note:** If your app targets Android 8.0 (API level 26) or higher, don't request the **`READ_SMS`** (https://developer.android.com/reference/android/Manifest.permission.html#READ_SMS) permission as part of verifying a user's credentials. Instead, generate an app-specific token using

`createAppSpecificSmsToken()` (https://developer.android.com/reference/android/telephony/SmsManager.html#createAppSpecificSmsToken(android.app.PendingIntent))
, then pass this token to another app or service that can send a verification SMS message.

Under certain conditions, it's also advantageous to let users know about sensitive data accesses in real-time. For example, if you're accessing the camera or microphone, it's usually a good idea to let the user know with a notification icon somewhere in your app, or in the notification tray (if the application is running in the background), so it doesn't seem like you're collecting data surreptitiously.

Ultimately, if you need to request a permission to make something in your app work, but the reason is not clear to the user, find a way to let the user know why you need the most sensitive permissions.

Test for both permissions models

Beginning with Android 6.0 (API level 23), users grant and revoke app permissions at run time, instead of doing so when they install the app. As a result, you'll have to test your app under a wider range of conditions. Prior to Android 6.0, you could reasonably assume that if your app is running at all, it has all the permissions it declares in the app manifest. Beginning with Android 6.0 the user can turn permissions on or off for *any* app, even an app that targets API level 22 or lower. You should test to ensure your app functions correctly whether or not it has any permissions.

The following tips will help you find permissions-related code problems on devices running API level 23 or higher:

- Identify your app's current permissions and the related code paths.
- Test user flows across permission-protected services and data.
- Test with various combinations of granted or revoked permissions. For example, a camera app might list `CAMERA` (https://developer.android.com/reference/android/Manifest.permission.html#CAMERA), `READ_CONTACTS` (https://developer.android.com/reference/android/Manifest.permission.html#READ_CONTACTS), and `ACCESS_FINE_LOCATION` (https://developer.android.com/reference/android/Manifest.permission.html#ACCESS_FINE_LOCATION) in its manifest. You should test the app with each of these permissions turned on and off, to make sure the app can handle all permission configurations gracefully.

- Use the adb (https://developer.android.com/tools/help/adb.html) tool to manage permissions from the command line:
  - List permissions and status by group:

    ```
    $ adb shell pm list permissions -d -g
    ```

  - Grant or revoke one or more permissions:

    ```
    $ adb shell pm [grant|revoke] <permission-name> ...
    ```

- Analyze your app for services that use permissions.

Additional resources

- Material Design guidelines for Android permissions (https://material.io/design/platform-guidance/android-permissions.html#usage)
- Android Marshmallow 6.0: Asking For Permission (https://www.youtube.com/watch?v=iZqDdvhTZj0): This video explains the Android runtime permission model and the right way to ask users for permissions.
- Explain why the app needs permissions (https://developer.android.com/training/permissions/requesting#explain)
- Best practices for unique identifiers (https://developer.android.com/training/articles/user-data-ids.html)

References

[1] *Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings*, by J. Lin B. Liu, N. Sadeh and J. Hong. In Proceedings of SOUPS 2014.

Previous
← Request app permissions (https://developer.android.com/training/permissions/requesting)

Next
Define custom permissions → (https://developer.android.com/guide/topics/permissions/defining)

Last updated June 15, 2018.

Twitter
Follow @AndroidDev on Twitter

Google+
Follow Android Developers on Google+

YouTube
Check out Android Developers on YouTube