Application Sandbox

Archived: 2026-04-05 17:56:52 UTC

The Android platform takes advantage of the Linux user-based protection to identify and isolate app resources. This isolates apps from each other and protects apps and the system from malicious apps. To do this, Android assigns a unique user ID (UID) to each Android app and runs it in its own process.

Android uses the UID to set up a kernel-level Application Sandbox. The kernel enforces security between apps and the system at the process level through standard Linux facilities such as user and group IDs that are assigned to apps. By default, apps can't interact with each other and have limited access to the OS. If app A tries to do something malicious, such as read app B's data or dial the phone without permission, it's prevented from doing so because it doesn't have the appropriate default user privileges. The sandbox is simple, auditable, and based on decades-old UNIX-style user separation of processes and file permissions.

Because the Application Sandbox is in the kernel, this security model extends to both native code and OS apps. All of the software above the kernel, such as OS libraries, app framework, app runtime, and all apps, run within the Application Sandbox. On some platforms, developers are constrained to a specific development framework, set of APIs, or language. On Android, there are no restrictions on how an app can be written that are required to enforce security; in this respect, native code is as sandboxed as interpreted code.

Protections

Generally, to break out of the Application Sandbox in a properly configured device, one must compromise the security of the Linux kernel. However, similar to other security features, individual protections enforcing the app sandbox are not invulnerable, so defense-in-depth is important to prevent single vulnerabilities from leading to compromise of the OS or other apps.

Android relies on a number of protections to enforce the app sandbox. These enforcements have been introduced over time and have significantly strengthened the original UID-based discretionary access control (DAC) sandbox. Previous Android releases included the following protections:

- In Android 5.0, SELinux provided mandatory access control (MAC) separation between the system and apps. However, all third-party apps ran within the same SELinux context so inter-app isolation was primarily enforced by UID DAC.
- In Android 6.0, the SELinux sandbox was extended to isolate apps across the per-physical-user boundary. In addition, Android also set safer defaults for app data: For apps with `targetSdkVersion >= 24`, default DAC permissions on an app's home dir changed from 751 to 700. This provided safer default for private app data (although apps can override these defaults).
- In Android 8.0, all apps were set to run with a `seccomp-bpf` filter that limited the syscalls that apps were allowed to use, thus strengthening the app/kernel boundary.
- In Android 9 all nonprivileged apps with `targetSdkVersion >= 28` must run in individual SELinux sandboxes, providing MAC on a per-app basis. This protection improves app separation, prevents

overriding safe defaults, and (most significantly) prevents apps from making their data world accessible.
- In Android 10 apps have a limited raw view of the filesystem, with no direct access to paths like /sdcard/DCIM. However, apps retain full raw access to their package-specific paths, as returned by any applicable methods, such as Context.getExternalFilesDir().

Guidelines for sharing files

Setting app data as world accessible is a poor security practice. Access is granted to everyone and it's not possible to limit access to only the intended recipient(s). This practice has led to information disclosure leaks and confused deputy vulnerabilities, and is a favorite target for malware that targets apps with sensitive data (such as email clients). In Android 9 and higher, sharing files this way is explicitly disallowed for apps with `targetSdkVersion>=28`.

Instead of making app data world-accessible, use the following guidelines when sharing files:

- If your app needs to share files with another app, use a content provider. Content providers share data with the proper granularity and without the many downsides of world-accessible UNIX permissions (for details, refer to Content provider basics).
- If your app has files that genuinely should be accessible to the world (such as photos), they must be media-specific (photos, videos, and audio files only) and stored using the MediaStore class. (For more details on how to add a media item, see Access media files from shared storage.)

The **Storage** runtime permission controls access to strongly-typed collections through **MediaStore**. For accessing weakly typed files such as PDFs and the MediaStore.Downloads class, apps must use intents like the `ACTION_OPEN_DOCUMENT` intent.

To enable Android 10 behavior, use the `requestLegacyExternalStorage` manifest attribute, and follow App permissions best practices.

- The manifest flag default value is `true` for apps targeting Android 9 and lower.
- The default value is false for apps targeting Android 10. To temporarily opt out of the filtered storage view in apps targeting Android 10, set the manifest flag's value to `true`.
- Using restricted permissions, the installer allowlists apps permitted for nonsandboxed storage. Nonallowlisted apps are sandboxed.
