Android Developers

# Android 7.0 Behavior Changes

Along with new features and capabilities, Android 7.0 includes a variety of system and API behavior changes. This document highlights some of the key changes that you should understand and account for in your apps.

If you have previously published an app for Android, be aware that your app might be affected by these changes in the platform.

In this document



## Battery and Memory

Android 7.0 includes system behavior changes aimed at improving the battery life of devices and reducing RAM usage. These changes can affect your app's access to system resources, along with the way your app interacts with other apps via certain implicit intents.

### Doze

Introduced in Android 6.0 (API level 23), Doze improves battery life by deferring CPU and network activities when a user leaves a device unplugged, stationary, and with the screen turned off. Android 7.0 brings further enhancements to Doze by applying a subset of CPU and network restrictions while the device is unplugged with the screen turned off, but not necessarily stationary, for example, when a handset is traveling in a user's pocket.

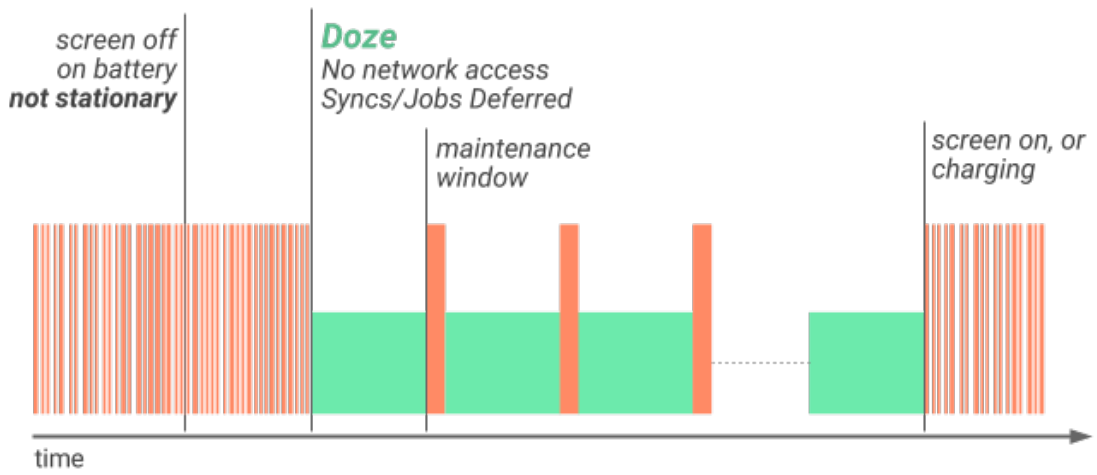


**Figure 1.** Illustration of how Doze applies a first level of system activity restrictions to improve battery life.

When a device is on battery power, and the screen has been off for a certain time, the device enters Doze and applies the first subset of restrictions: It shuts off app network access, and defers jobs and syncs. If the device is stationary for a certain time after entering Doze, the system applies the rest of the Doze restrictions to `PowerManager.WakeLock` (https://developer.android.com/reference/android/os/PowerManager.WakeLock.html), `AlarmManager` (https://developer.android.com/reference/android/app/AlarmManager.html) alarms, GPS, and Wi-Fi scans. Regardless of whether some or all Doze restrictions are being applied, the system wakes the device for brief maintenance windows, during which applications are allowed network access and can execute any deferred jobs/syncs.

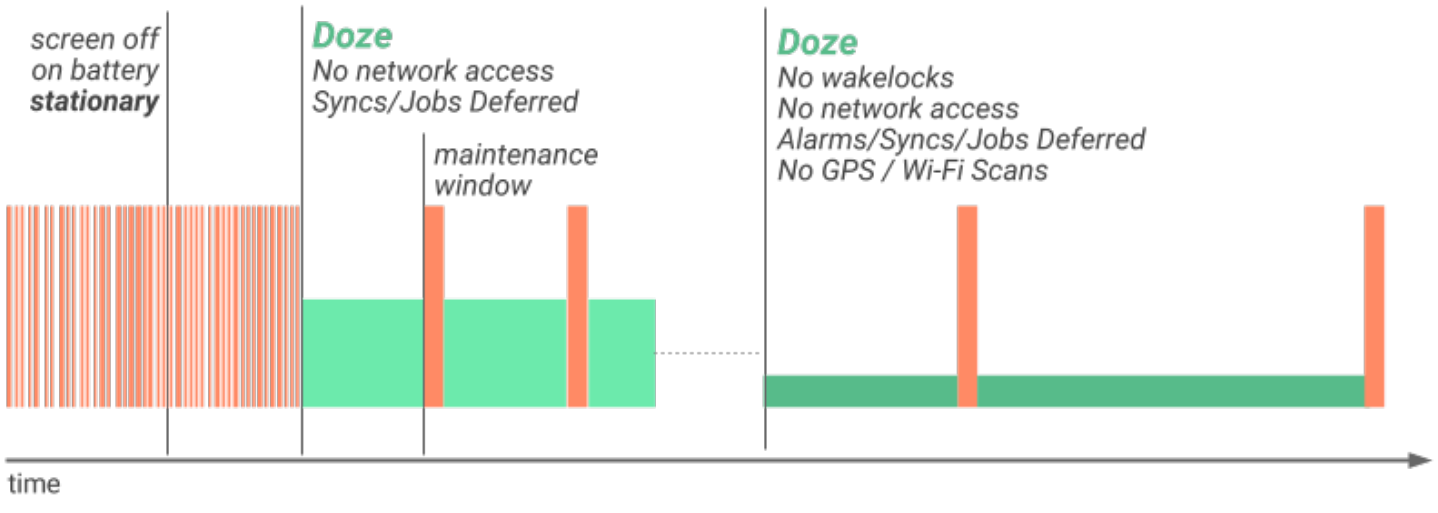


**Figure 2.** Illustration of how Doze applies a second level of system activity restrictions after the device is stationary for a certain time.

Note that activating the screen on or plugging in the device exits Doze and removes these processing restrictions. The additional behavior does not affect recommendations and best practices in adapting your app to the prior version of Doze introduced in Android 6.0 (API level 23), as discussed in Optimizing for Doze and App Standby (https://developer.android.com/training/monitoring-device-state/doze-standby.html). You should still follow those recommendations, such as using Google Cloud Messaging (GCM) to send and receive messages, and start planning updates to accommodate the additional Doze behavior.

## Project Svelte: Background Optimizations

Android 7.0 removes three implicit broadcasts in order to help optimize both memory use and power consumption. This change is necessary because implicit broadcasts frequently start apps that have registered to listen for them in the background. Removing these broadcasts can substantially benefit device performance and user experience.

Mobile devices experience frequent connectivity changes, such as when moving between Wi-Fi and mobile data. Currently, apps can monitor for changes in connectivity by registering a receiver for the implicit `CONNECTIVITY_ACTION` (https://developer.android.com/reference/android/net/ConnectivityManager.html#CONNECTIVITY_ACTION) broadcast in their manifest. Since many apps register to receive this broadcast, a single network switch can cause them all to wake up and process the broadcast at once.

Similarly, in previous versions of Android, apps could register to receive implicit `ACTION_NEW_PICTURE` (https://developer.android.com/reference/android/hardware/Camera.html#ACTION_NEW_PICTURE) and `ACTION_NEW_VIDEO` (https://developer.android.com/reference/android/hardware/Camera.html#ACTION_NEW_VIDEO) broadcasts from other apps, such as Camera. When a user takes a picture with the Camera app, these apps wake up to process the broadcast.

To alleviate these issues, Android 7.0 applies the following optimizations:

- Apps targeting Android 7.0 (API level 24) and higher do not receive `CONNECTIVITY_ACTION` (https://developer.android.com/reference/android/net/ConnectivityManager.html#CONNECTIVITY_ACTION) broadcasts if they declare their broadcast receiver in the manifest. Apps will still receive `CONNECTIVITY_ACTION` (https://developer.android.com/reference/android/net/ConnectivityManager.html#CONNECTIVITY_ACTION) broadcasts if they register their `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) with `Context.registerReceiver()` (https://developer.android.com/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)) and that context is still valid.
- The system no longer sends `ACTION_NEW_PICTURE` (https://developer.android.com/reference/android/hardware/Camera.html#ACTION_NEW_PICTURE) or `ACTION_NEW_VIDEO` (https://developer.android.com/reference/android/hardware/Camera.html#ACTION_NEW_VIDEO) broadcasts. This optimization affects all apps, not only those targeting Android 7.0.

If your app uses any of these intents, you should remove dependencies on them as soon as possible so that you can target Android 7.0 devices properly. The Android framework provides several solutions to mitigate the need for these implicit broadcasts. For example, the `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html) API provides a robust mechanism to schedule network operations when specified conditions, such as connection to an unmetered network, are met. You can even use `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html) to react to changes to content providers.

For more information about background optimizations in Android 7.0 (API level 24) and how to adapt your app, see Background Optimizations (https://developer.android.com/preview/features/background-optimization.html).

# Permissions Changes

Android 7.0 includes changes to permissions that may affect your app.

## File system permission changes

In order to improve the security of private files, the private directory of apps targeting Android 7.0 or higher has restricted access (`0700`). This setting prevents leakage of metadata of private files, such as their size or existence. This permission change has multiple side effects:

- Private files' file permissions should no longer be relaxed by the owner, and an attempt to do so using `MODE_WORLD_READABLE` (https://developer.android.com/reference/android/content/Context.html#MODE_WORLD_READABLE) and/or `MODE_WORLD_WRITEABLE` (https://developer.android.com/reference/android/content/Context.html#MODE_WORLD_WRITEABLE), will trigger a `SecurityException` (https://developer.android.com/reference/java/lang/SecurityException.html).

  > **Note:** As of yet, this restriction is not fully enforced. Apps may still modify permissions to their private directory using native APIs or the `File` (https://developer.android.com/reference/java/io/File.html) API. However, we strongly discourage relaxing the permissions to the private directory.

- Passing `file://` URIs outside the package domain may leave the receiver with an unaccessible path. Therefore, attempts to pass a `file://` URI trigger a `FileUriExposedException`. The recommended way to share the content of a private file is using the `FileProvider` (https://developer.android.com/reference/android/support/v4/content/FileProvider.html).

- The `DownloadManager` (https://developer.android.com/reference/android/app/DownloadManager.html) can no longer share privately stored files by filename. Legacy applications may end up with an unaccessible path when accessing `COLUMN_LOCAL_FILENAME` (https://developer.android.com/reference/android/app/DownloadManager.html#COLUMN_LOCAL_FILENAME). Apps targeting Android 7.0 or higher trigger a `SecurityException` (https://developer.android.com/reference/java/lang/SecurityException.html) when attempting to access `COLUMN_LOCAL_FILENAME` (https://developer.android.com/reference/android/app/DownloadManager.html#COLUMN_LOCAL_FILENAME). Legacy applications that set the download location to a public location by using `DownloadManager.Request.setDestinationInExternalFilesDir()` (https://developer.android.com/reference/android/app/DownloadManager.Request.html#setDestinationInExternalFilesDir(android.content.Context, java.lang.String, java.lang.String)) or `DownloadManager.Request.setDestinationInExternalPublicDir()` (https://developer.android.com/reference/android/app/DownloadManager.Request.html#setDestinationInExternalPublicDir(java.lang.String, java.lang.String)) can still access the path in `COLUMN_LOCAL_FILENAME` (https://developer.android.com/reference/android/app/DownloadManager.html#COLUMN_LOCAL_FILENAME), however, this method is strongly discouraged. The preferred way of accessing a file exposed by the `DownloadManager` (https://developer.android.com/reference/android/app/DownloadManager.html) is using `ContentResolver.openFileDescriptor()` (https://developer.android.com/reference/android/content/ContentResolver.html#openFileDescriptor(android.net.Uri, java.lang.String)).

# Sharing Files Between Apps

For apps targeting Android 7.0, the Android framework enforces the `StrictMode` (https://developer.android.com/reference/android/os/StrictMode.html) API policy that prohibits exposing `file://` URIs outside your app. If an intent containing a file URI leaves your app, the app fails with a `FileUriExposedException` exception.

To share files between applications, you should send a `content://` URI and grant a temporary access permission on the URI. The easiest way to grant this permission is by using the `FileProvider` (https://developer.android.com/reference/android/support/v4/content/FileProvider.html) class. For more information on permissions and sharing files, see Sharing Files (https://developer.android.com/training/secure-file-sharing/index.html).

# Accessibility Improvements

Android 7.0 includes changes intended to improve the usability of the platform for users with low or impaired vision. These changes should generally not require code changes in your app, however you should review these feature and test them with your app to assess potential impacts to user experience.

## Screen Zoom

Android 7.0 enables users to set **Display size** which magnifies or shrinks all elements on the screen, thereby improving device accessibility for users with low vision. Users cannot zoom the screen past a minimum screen width of sw320dp (http://developer.android.com/guide/topics/resources/providing-resources.html), which is the width of a Nexus 4, a common medium-sized phone.

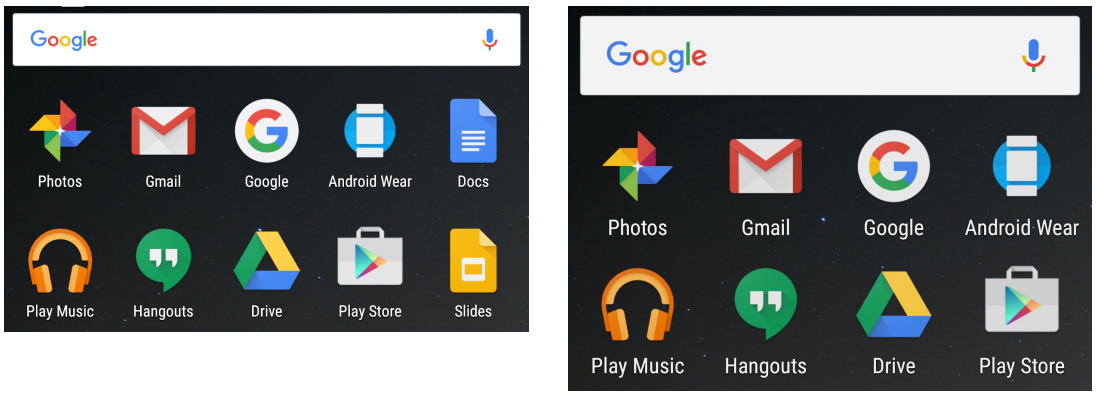


**Figure 3.** The screen on the right shows the effect of increasing the Display size of a device running an Android 7.0 system image.

When the device density changes, the system notifies running apps in the following ways:

- If an app targets API level 23 or lower, the system automatically kills all its background processes. This means that if a user switches away from such an app to open the *Settings* screen and changes the **Display size** setting, the system kills the app in the same manner that it would in a low-memory situation. If the app has any foreground processes, the system notifies those processes of the configuration change as described in Handling Runtime Changes (https://developer.android.com/guide/topics/resources/runtime-changes.html), just as if the device's orientation had changed.
- If an app targets Android 7.0, all of its processes (foreground and background) are notified of the configuration change as described in Handling Runtime Changes (https://developer.android.com/guide/topics/resources/runtime-changes.html).

Most apps do not need to make any changes to support this feature, provided the apps follow Android best practices. Specific things to check for:

- Test your app on a device with screen width `sw320dp` (`https://developer.android.com/guide/topics/resources/providing-resources.html`) and be sure it performs adequately.
- When the device configuration changes, update any density-dependent cached information, such as cached bitmaps or resources loaded from the network. Check for configuration changes when the app resumes from the paused state.

  > **Note:** If you cache configuration-dependent data, it's a good idea to include relevant metadata such as the appropriate screen size or pixel density for that data. Saving this metadata allows you to decide whether you need to refresh the cached data after a configuration change.

- Avoid specifying dimensions with px units, since they do not scale with screen density. Instead, specify dimensions with density-independent pixel (https://developer.android.com/guide/practices/screens_support.html) (`dp`) units.

## Vision Settings in Setup Wizard

Android 7.0 includes Vision Settings on the Welcome screen, where users can set up the following accessibility settings on a new device: **Magnification gesture**, **Font size**, **Display size** and **TalkBack**. This change increases the visibility of bugs related to different screen settings. To assess the impact of this feature, you should test your apps with these settings enabled. You can find the settings under **Settings > Accessibility**.

# NDK Apps Linking to Platform Libraries

Starting in Android 7.0, the system prevents apps from dynamically linking against non-NDK libraries, which may cause your app to crash. This change in behavior aims to create a consistent app experience across platform updates and

different devices. Even though your code might not be linking against private libraries, it's possible that a third-party static library in your app could be doing so. Therefore, all developers should check to make sure that their apps do not crash on devices running Android 7.0. If your app uses native code, you should only be using public NDK APIs (https://developer.android.com/ndk/guides/stable_apis.html).

There are three ways your app might be trying to access private platform APIs:

- Your app directly accesses private platform libraries. You should update your app to include its own copy of those libraries or use the public NDK APIs (https://developer.android.com/ndk/guides/stable_apis.html).
- Your app uses a third-party library that accesses private platform libraries. Even if you are certain your app doesn't access private libraries directly, you should still test your app for this scenario.
- Your app references a library that is not included in its APK. For example, this could happen if you tried to use your own copy of OpenSSL but forgot to bundle it with your app's APK. The app may run normally on versions of Android platform that includes `libcrypto.so`. However, the app could crash on later versions of Android that do not include this library (such as, Android 6.0 and later). To fix this, ensure that you bundle all your non-NDK libraries with your APK.

Apps should not use native libraries that are not included in the NDK because they may change or be removed between different versions of Android. The switch from OpenSSL to BoringSSL is an example of such a change. Also, because there are no compatibility requirements for platform libraries not included in the NDK, different devices may offer different levels of compatibility.

In order to reduce the impact that this restriction may have on currently released apps, a set of libraries that see significant use—such as `libandroid_runtime.so`, `libcutils.so`, `libcrypto.so`, and `libssl.so`—are temporarily accessible on Android 7.0 (API level 24) for apps targeting API level 23 or lower. If your app loads one of these libraries, logcat generates a warning and a toast appears on the target device to notify you. If you see these warnings, you should update your app to either include its own copy of those libraries or only use the public NDK APIs. Future releases of the Android platform may restrict the use of private libraries altogether and cause your app to crash.

All apps generate a runtime error when they call an API that is neither public nor temporarily accessible. The result is that `System.loadLibrary` and `dlopen(3)` both return `NULL`, and may cause your app to crash. You should review your app code to remove use of private platform APIs and thoroughly test your apps using a device or emulator running Android 7.0 (API level 24). If you are unsure whether your app uses private libraries, you can check logcat (#ndk-errors) to identify the runtime error.

The following table describes the behavior you should expect to see from an app depending on its use of private native libraries and its target API level (`android:targetSdkVersion`).

| Libraries | Target API level | Runtime access via dynamic linker | Android 7.0 (API level 24) behavior | Future Android platform behavior |
|---|---|---|---|---|
| NDK Public | Any | Accessible | Works as expected | Works as expected |
| Private (temporarily accessible private libraries) | 23 or lower | Temporarily accessible | Works as expected, but you receive a logcat warning. | Runtime error |
| Private (temporarily accessible private libraries) | 24 or higher | Restricted | Runtime error | Runtime error |
| Private (other) | Any | Restricted | Runtime error | Runtime error |

## Check if your app uses private libraries

To help you identify issues loading private libraries, logcat may generate a warning or runtime error. For example, if your app targets API level 23 or lower, and tries to access a private library on a device running Android 7.0, you may see a warning similar to the following:

```
03-21 17:07:51.502 31234 31234 W linker  : library "libandroid_runtime.so"
```

```
("/system/lib/libandroid_runtime.so") needed or dlopened by
"/data/app/com.popular-app.android-2/lib/arm/libapplib.so" is not accessible
for the namespace "classloader-namespace" - the access is temporarily granted
as a workaround for http://b/26394120
```

These logcat warnings tell you which which library is trying to access a private platform API, but will not cause your app to crash. If the app targets API level 24 or higher, however, logcat generates the following runtime error and your app may crash:

```
java.lang.UnsatisfiedLinkError: dlopen failed: library "libcutils.so"
("/system/lib/libcutils.so") needed or dlopened by
"/system/lib/libnativeloader.so" is not accessible for the namespace
"classloader-namespace"
  at java.lang.Runtime.loadLibrary0(Runtime.java:977)
  at java.lang.System.loadLibrary(System.java:1602)
```

You may also see these logcat outputs if your app uses third-party libraries that dynamically link to private platform APIs. The readelf tool in the Android 7.0DK allows you to generate a list of all dynamically linked shared libraries of a given `.so` file by running the following command:

```
aarch64-linux-android-readelf -dW libMyLibrary.so
```

## Update your app

Here are some steps you can take to fix these types of errors and make sure your app doesn't crash on future platform updates:

- If your app uses private platform libraries, you should update it to include its own copy of those libraries or use the public NDK APIs (https://developer.android.com/ndk/guides/stable_apis.html).
- If your app uses a third-party library that accesses private symbols, contact the library author to update the library.
- Make sure you package all your non-NDK libraries with your APK.
- Use standard JNI functions instead of `getJavaVM` and `getJNIEnv` from `libandroid_runtime.so`:

  ```
  AndroidRuntime::getJavaVM -> GetJavaVM from <jni.h>
  AndroidRuntime::getJNIEnv -> JavaVM::GetEnv or
  JavaVM::AttachCurrentThread from <jni.h>.
  ```

- Use `__system_property_get` instead of the private `property_get` symbol from `libcutils.so`. To do this, use `__system_property_get` with the following include:

  ```
  #include <sys/system_properties.h>
  ```

  > **Note:** The availability and contents of system properties is not tested through CTS. A better fix would be to avoid using these properties altogether.

- Use a local version of the `SSL_ctrl` symbol from `libcrypto.so`. For example, you should statically link `libcyrpto.a` in your `.so` file, or include a dynamically linked version of `libcrypto.so` from BoringSSL/OpenSSL and package it in your APK.

# Android for Work

Android 7.0 contains changes for apps that target Android for Work, including changes to certificate installation, password resetting, secondary user management, and access to device identifiers. If you are building apps for Android for Work environments, you should review these changes and modify your app accordingly.

- You must install a delegated certificate installer before the DPC can set it. For both profile and device-owner apps targeting Android 7.0 (API level 24), you should install the delegated certificate installer before the device policy controller (DPC) calls `DevicePolicyManager.setCertInstallerPackage()`. If the installer is not already installed, the system throws an `IllegalArgumentException`.
- Reset password restrictions for device admins now apply to profile owners. Device admins can no longer use `DevicePolicyManager.resetPassword()` to clear passwords or change ones that are already set. Device admins

can still set a password, but only when the device has no password, PIN, or pattern.

- Device and profile owners can manage accounts even if restrictions are set. Device owners and profile owners can call the Account Management APIs even if `DISALLOW_MODIFY_ACCOUNTS` user restrictions are in place.
- Device owners can manage secondary users more easily. When a device is running in device owner mode, the `DISALLOW_ADD_USER` restriction is automatically set. This prevents users from creating unmanaged secondary users. In addition, the `CreateUser()` and `createAndInitializeUser()` methods are deprecated; the new `DevicePolicyManager.createAndManageUser()` method replaces them.
- Device owners can access device identifiers. A Device owner can access the Wi-Fi MAC address of a device, using `DevicePolicyManagewr.getWifiMacAddress()`. If Wi-Fi has never been enabled on the device, this method returns a value of `null`.
- The Work Mode setting controls access to work apps. When work mode is off the system launcher indicates work apps are unavailable by greying them out. Enabling work mode again restores normal behavior.
- When installing a PKCS #12 file containing a client certificate chain and the corresponding private key from Settings UI, the CA certificate in the chain is no longer installed to the trusted credentials storage. This does not affect the result of `KeyChain.getCertificateChain()` (https://developer.android.com/reference/android/security/KeyChain.html#getCertificateChain(android.content.Context, java.lang.String)) when apps attempt to retrieve the client certificate chain later. If required, the CA certificate should be installed to the trusted credentials storage via Settings UI separately, with a DER-encoded format under a .crt or .cer file extension.
- Starting in Android 7.0, fingerprint enrollment and storage are managed per user. If a profile owner's Device Policy Client (DPC) targets API level 23 (or lower) on a device running Android 7.0 (API level 24), the user is still able to set fingerprint on the device, but work applications cannot access device fingerprint. When the DPC targets API level 24 and above, the user can set fingerprint specifically for work profile by going to **Settings > Security > Work profile security**.
- A new encryption status `ENCRYPTION_STATUS_ACTIVE_PER_USER` is returned by `DevicePolicyManager.getStorageEncryptionStatus()`, to indicate that encryption is active and the encryption key is tied to the user. The new status is only returned if DPC targets API Level 24 and above. For apps targeting earlier API levels, `ENCRYPTION_STATUS_ACTIVE` is returned, even if the encryption key is specific to the user or profile.
- In Android 7.0, several methods that would ordinarily affect the entire device behave differently if the device has a work profile installed with a separate work challenge. Rather than affecting the entire device, these methods apply only to the work profile. (The complete list of such methods is in the `DevicePolicyManager.getParentProfileInstance()` (https://developer.android.com/reference/android/app/admin/DevicePolicyManager.html#getParentProfileInstance(android.content.ComponentName)) documentation.) For example, `DevicePolicyManager.lockNow()` (https://developer.android.com/reference/android/app/admin/DevicePolicyManager.html#lockNow()) locks just the work profile, instead of locking the entire device. For each of these methods, you can get the old behavior by calling the method on the parent instance of the `DevicePolicyManager` (https://developer.android.com/reference/android/app/admin/DevicePolicyManager.html); you can get this parent by calling `DevicePolicyManager.getParentProfileInstance()` (https://developer.android.com/reference/android/app/admin/DevicePolicyManager.html#getParentProfileInstance(android.content.ComponentName)). So for example, if you call the parent instance's `lockNow()` (https://developer.android.com/reference/android/app/admin/DevicePolicyManager.html#lockNow()) method, the entire device is locked.

For more information about changes to Android for Work in Android 7.0, see Android for Work Updates (https://developer.android.com/preview/features/afw.html).

## Annotations Retention

Android 7.0 fixes a bug where the visibility of annotations was being ignored. This issue enabled the runtime to access annotations that it should not have been able to. These annotations included:

- `VISIBILITY_BUILD`: Intended to be visible only at build time.
- `VISIBILITY_SYSTEM`: Intended to be visible at runtime, but only to the underlying system.

If your app has relied on this behavior, please add a retention policy to annotations that must be available at runtime. You do so by using `@Retention(RetentionPolicy.RUNTIME)`.

## TLS/SSL Default Configuration Changes

Android 7.0 makes the following changes to the default TLS/SSL configuration used by apps for HTTPS and other TLS/SSL traffic:

- RC4 cipher suites are now disabled.
- CHACHA20-POLY1305 cipher suites are now enabled.

RC4 being disabled by default may lead to breakages in HTTPS or TLS/SSL connectivity when the server does not negotiate modern cipher suites. The preferred fix is to improve the server's configuration to enable stronger and more modern cipher suites and protocols. Ideally, TLSv1.2 and AES-GCM should be enabled, and Forward Secrecy cipher suites (ECDHE) should be enabled and preferred.

An alternative is to modify the app to use a custom `SSLSocketFactory` (https://developer.android.com/reference/javax/net/ssl/SSLSocketFactory.html) to communicate with the server. The factory should be designed to create `SSLSocket` (https://developer.android.com/reference/javax/net/ssl/SSLSocket.html) instances that have some of the cipher suites required by the server enabled in addition to default cipher suites.

> **Note:** These changes do not pertain to `WebView` (https://developer.android.com/reference/android/webkit/WebView.html).

## Other Important Points

- When an app is running on Android 7.0, but targets a lower API level, and the user changes display size, the app process is killed. The app must be able to gracefully handle this scenario. Otherwise, it crashes when the user restores it from Recents.

  You should test your app to ensure that this behavior does not occur. You can do so by causing an identical crash when killing the app manually via DDMS.

  Apps targeting Android 7.0 (API level 24) and above are not automatically killed on density changes; however, they may still respond poorly to configuration changes.

- Apps on Android 7.0 should be able to gracefully handle configuration changes, and should not crash on subsequent starts. You can verify app behavior by changing font size (**Setting** > **Display** > **Font size**), and then restoring the app from Recents.

- Due to a bug in previous versions of Android, the system did not flag writing to a TCP socket on the main thread as a strict-mode violation. Android 7.0 fixes this bug. Apps that exhibit this behavior now throw an `android.os.NetworkOnMainThreadException`. Generally, performing network operations on the main thread is a bad idea because these operations usually have a high latency that causes ANRs and jank.

- The `Debug.startMethodTracing()` family of methods now defaults to storing output in your package-specific directory on shared storage, instead of at the top level of the SD card. This means apps no longer need to request the `WRITE_EXTERNAL_STORAGE` permission to use these APIs.

- Many platform APIs have now started checking for large payloads being sent across `Binder` (https://developer.android.com/reference/android/os/Binder.html) transactions, and the system now rethrows `TransactionTooLargeExceptions` as `RuntimeExceptions`, instead of silently logging or suppressing them. One common example is storing too much data in `Activity.onSaveInstanceState()` (https://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle)), which causes `ActivityThread.StopInfo` to throw a `RuntimeException` when your app targets Android 7.0.

- If an app posts `Runnable` (https://developer.android.com/reference/java/lang/Runnable.html) tasks to a `View` (https://developer.android.com/reference/android/view/View.html), and the `View` (https://developer.android.com/reference/android/view/View.html) is not attached to a window, the system queues the `Runnable` (https://developer.android.com/reference/java/lang/Runnable.html) task with the `View` (https://developer.android.com/reference/android/view/View.html); the `Runnable` (https://developer.android.com/reference/java/lang/Runnable.html) task does not execute until the `View` (https://developer.android.com/reference/android/view/View.html) is attached to a window. This behavior fixes the following bugs:

    - If an app posted to a `View` (https://developer.android.com/reference/android/view/View.html) from a thread other than the intended window's UI thread, the `Runnable` (https://developer.android.com/reference/java/lang/Runnable.html) may run on the wrong thread as a result.
    - If the `Runnable` (https://developer.android.com/reference/java/lang/Runnable.html) task was posted from a thread other than a looper thread, the app could expose the `Runnable` (https://developer.android.com/reference/java/lang/Runnable.html) task.
- If an app on Android 7.0 with `DELETE_PACKAGES` (https://developer.android.com/reference/android/Manifest.permission.html#DELETE_PACKAGES) permission tries to delete a package, but a different app had installed that package, the system requires user confirmation. In this scenario, apps should expect `STATUS_PENDING_USER_ACTION` (https://developer.android.com/reference/android/content/pm/PackageInstaller.html#STATUS_PENDING_USER_ACTION) as the return status when they invoke `PackageInstaller.uninstall()` (https://developer.android.com/reference/android/content/pm/PackageInstaller.html#uninstall(android.content.pm.VersionedPackage, android.content.IntentSender)).
- The JCA provider called *Crypto* is deprecated, because its only algorithm, SHA1PRNG, is cryptographically weak. Apps can no longer use SHA1PRNG to (insecurely) derive keys, because this provider is no longer available. For more information, see the blog post Security "Crypto" provider deprecated in Android N (http://android-developers.blogspot.com/2016/06/security-crypto-provider-deprecated-in.html) .