Behavior changes: all apps

Archived: 2026-04-06 00:59:31 UTC

Android 9 (API level 28) introduces a number of changes to the Android system. The following behavior changes apply to *all apps* when they run on the Android 9 platform, regardless of the API level that they are targeting. All developers should review these changes and modify their apps to support them properly, where applicable to the app.

For changes that only affect apps that target API level 28 or higher, see Behavior changes: apps targeting API Level 28+.

Power management

Android 9 introduces new features to improve device power management. These changes, along with features that were already present before Android 9, help to ensure that system resources are made available to the apps that need them the most.

For details, see Power management.

Privacy changes

To enhance user privacy, Android 9 introduces several behavior changes, such as limiting background apps' access to device sensors, restricting information retrieved from Wi-Fi scans, and new permission rules and permission groups related to phone calls, phone state, and Wi-Fi scans.

These changes affect all apps running on Android 9, regardless of target SDK version.

Limited access to sensors in background

Android 9 limits the ability for background apps to access user input and sensor data. If your app is running in the background on a device running Android 9, the system applies the following restrictions to your app:

- Your app cannot access the microphone or camera.
- Sensors that use the continuous reporting mode, such as accelerometers and gyroscopes, don't receive events.
- Sensors that use the on-change or one-shot reporting modes don't receive events.

If your app needs to detect sensor events on devices running Android 9, use a foreground service.

Restricted access to call logs

Android 9 introduces the `CALL_LOG` permission group and moves the `READ_CALL_LOG`, `WRITE_CALL_LOG`, and `PROCESS_OUTGOING_CALLS` permissions into this group. In previous versions of Android, these permissions were

located in the `PHONE` permission group.

This `CALL_LOG` permission group gives users better control and visibility to apps that need access to sensitive information about phone calls, such as reading phone call records and identifying phone numbers.

If your app requires access to call logs or needs to process outgoing calls, you must explicitly request these permissions from the `CALL_LOG` permission group. Otherwise, a `SecurityException` occurs.

Note: Because these permissions have changed groups and are granted at runtime, it's possible for the user to deny your app access to phone call logs information. In this case, your app should be able to handle the lack of access to information gracefully.

If your app is already following runtime permissions best practices, it can handle the change in permission group.

Restricted access to phone numbers

Apps running on Android 9 cannot read phone numbers or phone state without first acquiring the `READ_CALL_LOG` permission, in addition to the other permissions that your app's use cases require.

Phone numbers associated with incoming and outgoing calls are visible in the phone state broadcast, such as for incoming and outgoing calls and are accessible from the `PhoneStateListener` class. Without the `READ_CALL_LOG` permission, however, the phone number field that's provided in `PHONE_STATE_CHANGED` broadcasts and through `PhoneStateListener` is empty.

To read phone numbers from phone state, update your app to request the necessary permissions based on your use case:

- To read numbers from the `PHONE_STATE` intent action, you need both the `READ_CALL_LOG` permission and the `READ_PHONE_STATE` permission.
- To read numbers from `onCallStateChanged()`, you need the `READ_CALL_LOG` permission only. You don't need the `READ_PHONE_STATE` permission.

Restricted access to Wi-Fi location and connection information

In Android 9, the permission requirements for an app to perform Wi-Fi scans are more strict than in previous versions. For details, see Wi-Fi scanning restrictions.

Similar restrictions also apply to the `getConnectionInfo()` method, which returns a `WifiInfo` object describing the current Wi-Fi connection. You can only use this object's methods to retrieve SSID and BSSID values if the calling app has the following permissions:

- ACCESS_FINE_LOCATION **or** ACCESS_COARSE_LOCATION
- ACCESS_WIFI_STATE

Retrieving the SSID or BSSID also requires location services to be enabled on the device (under **Settings > Location**).

Information removed from Wi-Fi service methods

In Android 9, the following events and broadcasts don't receive information about the user's location or personally identifiable data:

- The `getScanResults()` and `getConnectionInfo()` methods from `WifiManager`.
- The `discoverServices()` and `addServiceRequest()` methods from `WifiP2pManager`.
- The `NETWORK_STATE_CHANGED_ACTION` broadcast.

The `NETWORK_STATE_CHANGED_ACTION` system broadcast from Wi-Fi no longer contains SSID (previously EXTRA_SSID), BSSID (previously EXTRA_BSSID), or connection information (previously EXTRA_NETWORK_INFO). If your app needs this information, call `getConnectionInfo()` instead.

Telephony information now relies on device location setting

If the user has disabled device location on a device running Android 9, the following methods don't provide results:

- `getAllCellInfo()`
- `listen()`
- `getCellLocation()`
- `getNeighboringCellInfo()`

Restrictions on use of non-SDK interfaces

To help ensure app stability and compatibility, the platform restricts the use of some non-SDK methods and fields; these restrictions apply whether you attempt to access these methods and fields directly, via reflection, or using JNI. In Android 9, your app can continue to access these restricted interfaces; the platform uses toasts and log entries to bring them to your attention. If your app shows such a toast, it is important that you pursue an implementation strategy other than the restricted interface. If you feel that no alternative strategy is feasible, you may file a bug to request reconsideration of the restriction.

Restrictions on Non-SDK Interfaces contains further important information. You should review it to ensure that your app continues to function properly.

Security behavior changes

Device security changes

Android 9 adds several capabilities that improve your app's security, regardless of which version your app targets.

TLS implementation changes

The system's TLS implementation has undergone several changes in Android 9:

- If an instance of `SSLSocket` fails to connect while it's being created, the system throws an `IOException` instead of a `NullPointerException`.
- The `SSLEngine` class cleanly handles any `close_notify` alerts that occur.

To learn more about making secure web requests in an Android app, see An HTTPS example.

Stricter SECCOMP filter

Android 9 further restricts the system calls that are available to apps. This behavior is an extension of the SECCOMP filter that Android 8.0 (API level 26) includes.

Cryptographic changes

Android 9 introduces several changes to the implementation and handling of cryptographic algorithms.

Conscrypt implementations of parameters and algorithms

Android 9 provides additional implementations of algorithm parameters in Conscrypt. These parameters include: AES, DESEDE, OAEP, and EC. The Bouncy Castle versions of these parameters and many algorithms have been deprecated as of Android 9.

If your app targets Android 8.1 (API level 27) or lower, you receive a warning when requesting the Bouncy Castle implementation of one of these deprecated algorithms. If you target Android 9, however, these requests each throw a `NoSuchAlgorithmException`.

Other changes

Android 9 introduces several other changes related to cryptography:

- When using PBE keys, if Bouncy Castle is expecting an initialization vector (IV) and your app doesn't supply one, you receive a warning.
- The Conscrypt implementation of the ARC4 cipher allows you to specify either **ARC4/ECB/NoPadding** or **ARC4/NONE/NoPadding**.
- The Crypto Java Cryptography Architecture (JCA) provider has been removed. As a result, if your app calls `SecureRandom.getInstance("SHA1PRNG", "Crypto")`, a `NoSuchProviderException` occurs.
- If your app parses RSA keys from buffers that are larger than the key structure, an exception no longer occurs.

To learn more about using Android's cryptographic capabilities, see Cryptography.

Android secure encrypted files are no longer supported

Android 9 completely removes support for Android secure encrypted files (ASECs).

In Android 2.2 (API level 8), Android introduced ASECs to support apps-on-SD-card functionality. On Android 6.0 (API level 23), the platform introduced an adoptable storage device technology that developers can use in

place of ASECs.

Updates to the ICU libraries

Android 9 uses version 60 of the ICU library. Android 8.0 (API level 26) and Android 8.1 (API level 27) use ICU 58.

ICU is used to provide public APIs beneath the `android.icu package` and is used internally in the Android platform for internationalization support. For example, it is used to implement Android classes in `java.util`, `java.text`, and `android.text.format`.

The update to ICU 60 contains many small but useful changes, including Emoji 5.0 data support and improved date/time formats, as documented in the ICU 59 and ICU 60 release notes.

Notable changes in this update:

- The way the platform handles time zones has changed.
    - The platform handles GMT and UTC better; UTC is no longer a synonym for GMT.

      ICU now provides translated zone names for GMT and UTC. This change affects `android.icu` formatting and parsing behavior for zones like "GMT", "Etc/GMT", "UTC", "Etc/UTC", and "Zulu".

    - `java.text.SimpleDateFormat` now uses ICU to provide display names for UTC /GMT, meaning:
        - Formatting `zzzz` generates a long localized string for many locales. Previously, it produced "UTC" for UTC and strings like "GMT+00:00" for GMT.
        - Parsing `zzzz` recognizes strings like "Universal Coordinated Time", and "Greenwich Mean Time".
        - Apps may encounter compatibility problems if they assume that "UTC" or "GMT+00:00" are output for `zzzz` in all languages.
    - The behavior of `java.text.DateFormatSymbols.getZoneStrings()` has changed:
        - As with `SimpleDateFormat`, UTC and GMT now have long names. DST variants of time zone names for the UTC zone, such as "UTC", "Etc/UTC", and "Zulu", become GMT+00:00, which is the standard fallback when there are no names available, rather than the hard-coded string `UTC`.
        - Some zone IDs are correctly recognized as synonyms for other zones, so that Android finds strings for archaic zone IDs, such as `Eire`, that previously could not be resolved.
    - Asia/Hanoi is no longer a recognized zone. For this reason `java.util.TimeZones.getAvailableIds()` does not return this value, and `java.util.TimeZone.getTimeZone()` does not recognize it. This behavior is consistent with the existing `android.icu` behavior.
- The `android.icu.text.NumberFormat.getInstance(ULocale, PLURALCURRENCYSTYLE).parse(String)` method may throw a `ParseException` even when parsing legitimate currency text. Avoid this problem by using `NumberFormat.parseCurrency`, available since Android 7.0 (API level 24), for `PLURALCURRENCYSTYLE`-style currency text.

Android Test changes

Android 9 introduces several changes to the Android Test framework's library and class structure. These changes help developers use framework-supported, public APIs, but the changes also allow for more flexibility in building and running tests using third-party libraries or custom logic.

Libraries removed from framework

Android 9 reorganizes the JUnit-based classes into three libraries: **android.test.base**, **android.test.runner**, and **android.test.mock**. This change allows you to run tests against a version of JUnit that works best with your project's dependencies. This version of JUnit might be different than the one that `android.jar` provides.

To learn more about how the JUnit-based classes are organized into these libraries, as well as how to prepare your app's project for writing and running tests, see Set up project for Android Test.

Test suite build changes

The `addRequirements()` method in the `TestSuiteBuilder` class has been removed, and the `TestSuiteBuilder` class itself been deprecated. The `addRequirements()` method had required developers to supply arguments whose types are hidden APIs, making the API invalid.

Java UTF decoder

UTF-8 is the default charset in Android. A UTF-8 byte sequence can be decoded by a `String` constructor, such as `String(byte[] bytes)`.

The UTF-8 decoder in Android 9 follows the Unicode standards more strictly than in previous versions. The changes include the following:

- The non-shortest form of UTF-8, such as `<C0, AF>`, is treated as ill-formed.
- The surrogate form of UTF-8, such as `U+D800` .. `U+DFFF`, is treated as ill-formed.
- The maximal subpart is replaced by a single `U+FFFD`. For example, in the byte sequence "`41 C0 AF 41 F4 80 80 41`," the maximal subparts are "`C0`," "`AF`," and "`F4 80 80`." "`F4 80 80`" can be the initial subsequence of "`F4 80 80 80`", but "`C0`" can't be the initial subsequence of any well-formed code unit sequence. Thus, the output should be "`A\ufffd\ufffdA\ufffdA`."
- To decode a modified UTF-8 / CESU-8 sequence in Android 9 or higher, use the `DataInputStream.readUTF()` method or the `NewStringUTF()` JNI method.

Hostname verification using a certificate

RFC 2818 describes two methods to match a domain name against a certificate—using the available names within the `subjectAltName` (`SAN`) extension, or in the absence of a `SAN` extension, falling back to the `commonName` (`CN`).

However, the fallback to the `CN` was deprecated in RFC 2818. For this reason, Android no longer falls back to using the `CN`. To verify a hostname, the server must present a certificate with a matching `SAN`. Certificates that don't contain a `SAN` matching the hostname are no longer trusted.

Network address lookups can cause network violations

Network address lookups that require name resolution can involve network I/O and are therefore considered blocking operations. Blocking operations on the main thread can cause pauses or jank.

The `StrictMode` class is a development tool that helps developers to detect problems in their code.

In Android 9 and higher, `StrictMode` detects network violations caused by network address lookups that require name resolution.

You should not ship your apps with `StrictMode` enabled. If you do, then your apps can experience exceptions, such as `NetworkOnMainThreadException` when using the `detectNetwork()` or `detectAll()` methods to get a policy that detects network violations.

Resolving a numeric IP address isn't considered a blocking operation. Numeric IP address resolution works the same as in versions before Android 9.

Socket tagging

In platform versions lower than Android 9, if a socket is tagged using the `setThreadStatsTag()` method, the socket is untagged when it's sent to another process using binder IPC with a `ParcelFileDescriptor` container.

In Android 9 and higher, the socket tag is kept when it’s sent to another process using binder IPC. This change can affect network traffic statistics, for example, when using the `queryDetailsForUidTag()` method.

If you want to retain the behavior of the previous versions, which untags a socket that is sent to another process, you can call `untagSocket()` before sending the socket.

Reported amount of available bytes in socket

The `available()` method returns `0` when it's called after invoking the `shutdownInput()` method.

More detailed network capabilities reporting for VPNs

In Android 8.1 (API level 27) and lower, the `NetworkCapabilities` class only reported a limited set of information for VPNs, such as `TRANSPORT_VPN`, but omitting `NET_CAPABILITY_NOT_VPN`. This limited information made it difficult to determine if using a VPN would result in charges to the app's user. For example, checking `NET_CAPABILITY_NOT_METERED` would not determine whether the underlying networks were metered or not.

In Android 9 and higher, when a VPN calls the `setUnderlyingNetworks()` method, the Android system merges the transports and capabilities of any underlying networks and returns the result as the effective network capabilities of the VPN network.

In Android 9 and higher, apps that already check for `NET_CAPABILITY_NOT_METERED` receive the network capabilities of the VPN and the underlying networks.

Files in xt_qtaguid folder are no longer available to apps

Beginning with Android 9, apps are not allowed to have direct read access to files in the `/proc/net/xt_qtaguid` folder. The reason is to ensure consistency with some devices that don't have these files at all.

The public APIs that rely on these files, `TrafficStats` and `NetworkStatsManager`, continue to work as intended. However, the unsupported cutils functions, such as `qtaguid_tagSocket()`, may not work as expected—or at all— on different devices.

FLAG_ACTIVITY_NEW_TASK requirement is now enforced

With Android 9, you cannot start an activity from a non-activity context unless you pass the intent flag `FLAG_ACTIVITY_NEW_TASK`. If you attempt to start an activity without passing this flag, the activity does not start, and the system prints a message to the log.

Screen rotation changes

Beginning with Android 9, there are significant changes to the **portrait** rotation mode. In Android 8.0 (API level 26), users could toggle between **auto-rotate** and **portrait** rotation modes using a Quicksettings tile or Display settings. The portrait mode has been renamed **rotation lock** and it is active when auto-rotate is toggled off. There are no changes to auto-rotate mode.

When the device is in rotation lock mode, users can lock their screen to any rotation supported by the top, visible Activity. An Activity should not assume it will always be rendered in portrait. If the top Activity can be rendered in multiple rotations in auto-rotate mode, the same options should be available in rotation locked mode, with some exceptions based on the Activity's `screenOrientation` setting (see the table below).

Activities that request a specific orientation (for example, `screenOrientation=landscape`) ignore the user lock preference and behave the same way as in Android 8.0.

The screen orientation preference can be set at the Activity level in the Android Manifest, or programmatically with setRequestedOrientation().

The rotation lock mode works by setting the user rotation preference which the WindowManager uses when handling Activity rotation. The user rotation preference might be changed in the following cases. Note that there is a bias to return to the device's natural rotation, which is normally portrait for phone form factor devices:

- When the user accepts a rotation suggestion the rotation preference changes to the suggestion.
- When the user switches to a forced portrait app (including the lock screen or the launcher) the rotation preference changes to portrait.

The following table summarizes rotation behavior for the common screen orientations:

Screen Orientation	Behavior
unspecified, user	In auto-rotate and rotation lock the Activity can be rendered in portrait or landscape (and the reverse). Expect to support both portrait and landscape layouts.
userLandscape	In auto-rotate and rotation lock the Activity can be rendered in either landscape or reverse landscape. Expect to support only landscape layouts.
userPortrait	In auto-rotate and rotation lock the Activity can be rendered in either portrait or reverse portrait. Expect to support only portrait layouts.
fullUser	In auto-rotate and rotation lock the Activity can be rendered in portrait or landscape (and the reverse). Expect to support both portrait and landscape layouts. Rotation lock users will be given the option to lock to reverse portrait, often 180º.
sensor, fullSensor, sensorPortrait, sensorLandscape	The rotation lock mode preference is ignored and is treated as if auto-rotate is active. Only use this in exceptional circumstances with very careful UX consideration.

Apache HTTP client deprecation affects apps with non-standard ClassLoader

With Android 6.0, we removed support for the Apache HTTP client. This change has no effect on the great majority of apps that do not target Android 9 or higher. However, the change can affect certain apps that use a nonstandard `ClassLoader` structure, even if the apps do not target Android 9 or higher.

An app can be affected if it uses a non-standard `ClassLoader` that explicitly delegates to the system `ClassLoader`. These apps need to delegate to the *app* `ClassLoader` instead when looking for classes in `org.apache.http.*`. If they delegate to the system `ClassLoader`, the apps will fail on Android 9 or higher with a `NoClassDefFoundError`, because those classes are no longer known to the system `ClassLoader`. To prevent similar problems in the future, apps should in general load classes through the app `ClassLoader` rather than accessing the system `ClassLoader` directly.

Enumerating cameras

Apps running on Android 9 devices can discover every available camera by calling `getCameraIdList()`. An app should not assume that the device has only a single back camera or only a single front camera.

For example, if your app has a button to switch between the front and back cameras, there may be more than one front or back camera to choose from. You should walk the camera list, examine each camera's characteristics, and decide which cameras to expose to the user.

Source: https://developer.android.com/about/versions/pie/android-9.0-changes-all#bg-sensor-access