(Un)protected Broadcasts in Android 9 and 10

Ryan Johnson, Mohamed Elsabagh, & Angelos Stavrou

Dec. 21st, 2020

Executive Summary

We discovered a systemic vulnerability affecting Android version 9, Android version 10, and Android version 11 Developer Preview that allowed third-party apps to spoof certain protected broadcast Intent messages, allowing the sending of unauthorized messages that only the Android system and privileged pre-installed apps should be authorized to send. This *(un)protected broadcast vulnerability* occurs when an app declares that the system must protect some broadcast Intent message from being sent by other apps, yet --- due to a bug in Android Open Source Project (AOSP) --- the system granted that protection only to apps installed at a specific location on the file system. In other words, unless the app is installed at a certain path on the file system, the system would silently not honor these protection requests, leaving the app's broadcast messages unprotected at runtime.

Specifically, on vulnerable versions of Android, only pre-installed apps that reside in a `priv-app` directory (e.g., `/system/priv-app/SystemUI/SystemUI.apk`) can register protected broadcasts with the system. This leaves apps that are not present in a `priv-app` directory (e.g., other pre-installed apps or third party apps installed from the market) unable to have their protected broadcast declarations honored by the system which provides no access control and allows them to be sent by any app co-located on the device.

The lack of protection of protected broadcast Intent messages enables unauthorized parties to escalate their privileges where they can send spoofed messages to carry out functionalities they do not have the capability or authorization to perform. This can be viewed as a *confused deputy* problem since the process (deputy) receiving the broadcast Intent message acts upon it as if it was from an authorized source. We identified numerous Android vendors and devices that are impacted by this vulnerability where unauthorized apps can exploit vulnerable pre-installed apps to perform highly privileged functionalities, including arbitrary command execution with system privileges, access to the logcat log, and access to Personally Identifiable Information (PII).

1. Background

1.1 Broadcast Intent Messages

In Android, an Intent is a message used by Android app components to share data and events or request actions from other components.[1] An Intent message will generally be delivered to a single recipient component, except for broadcast Intent messages which can be delivered to multiple recipients.

A broadcast Intent typically contains a string indicating an event or action being announced or requested. This string

[1] https://developer.android.com/reference/android/content/Intent

is referred to as an “action string” in the Android world. Apps intending to receive broadcast messages can register handlers (called broadcast receivers) to receive and process broadcast messages. Broadcast Intents can also contain the exact address of the recipient component (typically the recipient app package name and a fully-qualified component class name) in which case the system would only unicast them to that specific recipient. Listing 1 shows an example declaration of a broadcast receiver in the pre-installed Phone app that registered to receive the `BOOT_COMPLETED` broadcast action that is sent when the system finishes booting up.

```
<receiver android:name="OtaStartupReceiver" ...>
    <intent-filter ...>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

Listing 1. A Broadcast Receiver declaration in the `AndroidManifest.xml` file in the Phone app.

If a third-party app can send broadcast Intents with an action string of `android.intent.action.BOOT_COMPLETED`, then it could potentially cause data corruption or data loss by causing initialization routines to execute in the receiving apps on the device by making it appear that the device has just finished the boot process even though it hasn’t.

1.2. Protected Broadcasts

There is little available in regard to documentation on the use of the protected broadcast primitive. By reviewing Android Open Source Project (AOSP) source code and experimenting with Android devices, we discovered what entities could declare protected broadcasts and also what entities could send them.

Based on how this protected broadcast primitive was used by pre-installed apps and the AOSP source code, we found that the Android Framework can prevent unauthorized apps from sending specific broadcast action strings if these action strings are declared via a `protected-broadcast` element in an `AndroidManifest.xml` file of an app or of the Android Framework itself. The AOSP Android Framework declares numerous protected broadcasts in its `AndroidManifest.xml` file. Vendors tend to also declare protected actions using the protected broadcast primitive and sometimes they add their own framework Android Package (APK) to protect additional broadcasts. A concrete example from the `AndroidManifest.xml` file of the Android Framework is provided in Listing 2.[2]

```
<protected-broadcast android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
<protected-broadcast android:name="android.intent.action.BOOT_COMPLETED"/>
<protected-broadcast android:name="android.intent.action.LOCALE_CHANGED"/>
```

Listing 2. Declarations of protected broadcasts.

The protected broadcast primitive instructs the system to perform access control by limiting which processes can send broadcast Intents containing these action strings. For example, if a third-party app attempts to send a broadcast Intent with an action string of `android.net.conn.CONNECTIVITY_CHANGE`, then the system will block the request and throw a `java.lang.SecurityException` in the process context of the caller app. Listing 3 provides a source code snippet from AOSP Android 9.0 release code where the `ActivtiyManagerService` evaluates whether the sender can send a broadcast Intent with a protected action.[3] Listing 3 shows that the sending of a broadcast Intent

[2] The Android Framework APK generally has a path of `/system/framework/framework-res.apk`.

[3] https://android.googlesource.com/platform/frameworks/base/+/00d9027/services/core/java/com/android/server/am/ActivityManagerService.java#21297

with that action string is restricted to critical processes executing with hard-coded system User IDs (e.g., `root`, `system`, `phone`, `nfc`, `bluetooth`, etc.) and persistent pre-installed apps (e.g., the Phone app). Otherwise, a `SecurityException` will be thrown in the caller.

```
final boolean isCallerSystem;
switch (UserHandle.getAppId(callingUid)) {
   case ROOT_UID:
   case SYSTEM_UID:
   case PHONE_UID:
   case BLUETOOTH_UID:
   case NFC_UID:
   case SE_UID:
       isCallerSystem = true;
       break;
   default:
       isCallerSystem = (callerApp != null) && callerApp.persistent;
       break;
}
// First line security check before anything else: stop non-system apps from
// sending protected broadcasts.
if (!isCallerSystem) {
   if (isProtectedBroadcast) {
       String msg = "Permission Denial: not allowed to send broadcast "
               + action + " from pid="
               + callingPid + ", uid=" + callingUid;
       Slog.w(TAG, msg);
       throw new SecurityException(msg);
   } ...
```

Listing 3. AOSP Android 9.0 Logic for Restricting the Sending of Protected Broadcasts.

1.3 The (Un)protected Broadcast Vulnerability

Support for the protected-broadcast primitive appears to go all the way back to very early days of Android. However, we noticed that there was a change in behavior starting with Android 9.0 where the system enforced protected broadcasts only for apps residing in specific directories on the filesystem. In the general case on Android 9.0 and above, apps that reside in the `/system/framework` directory and `priv-app` directories (e.g., `/system/priv-app`) are granted this protection, **whereas apps that reside elsewhere (e.g., `/system/app`) are not.** The protected-broadcast declaration for these apps that are not in a protected location is silently ignored by the system at runtime, leaving the actions (unexpectedly) unprotected. This allows third-party apps to elevate their privileges by sending spoofed broadcast intents using these unprotected actions to unsuspecting apps, crossing security boundaries.

The vulnerability was present in AOSP code that affected Android devices running version Android 9.0 and Android 10. At the time we reported the vulnerability, it was present in Android 11 Developer Preview 3 code and in the `master` branch of AOSP code (vulnerable AOSP versions have been patched as of the time of this writing). Android vendors use AOSP code for a particular Android version and then customize it to differentiate themselves amongst other Android vendors with the aim of giving themselves a competitive advantage. Any Android vendors, including Google, that use Android 9 and 10 for a specific build are vulnerable to the *(un)protected broadcast vulnerability*, although the scope and impact of exploiting the vulnerability depends on the specific functionalities of impacted apps declaring protected actions that will not actually be protected at runtime

1.4 Public Disclosure Timeline

We responsibly disclosed this vulnerability to Google through their bug reporting system and they internally tracked the vulnerability with an ID of A-158570769. Google classified the vulnerability as a high impact escalation of privilege vulnerability in their September 2020 Android Security Bulletin.[4] The *(un)protected broadcast vulnerability* was granted a Common Vulnerabilities and Exposures (CVE) ID of CVE-2020-0391 with a Common Vulnerability Scoring System (CVSS) v3 base score of 7.8 (high).[5] The git commit that fixed the vulnerability is provided here.[6] Examining the commit, the change to fix the vulnerability is to offer protection to the broadcast actions of all pre-installed apps regardless of their location on the filesystem.

- 05/08/2020: Initial disclosure to Android Security Team and affected vendors.
- 06/08/2020: Submitted vulnerability report to Google's IssueTracker.
- 06/09/2020: Submission acknowledged.
- 06/18/2020: Google finished their initial assessment and ranked the severity as "High".
- 08/21/2020: Google assigned CVE-2020-0391 for the vulnerability.
- 09/08/2020: Google changed the vulnerability status to "fixed" and provided a bug bounty.

2. Vulnerability Technical Details

The vulnerability was first introduced in the initial release of Android Pie (9.0) AOSP code and existed in the `master` branch of AOSP at the time we informed Google on May 8th, 2020. In Android Oreo (8.0), any pre-installed app can declare protected broadcast actions in their respective `AndroidManifest.xml` file and they will be registered by the system (i.e., `PackageManagerService`) as a protected broadcast during system startup. With the introduction of Android Pie, additional measures were taken to differentiate pre-installed apps with regards to their capabilities on the system. One specific differentiation between apps is determined by the path of a pre-installed app's APK file on the device. Starting with Android Pie, only APKs contained in the following directories can successfully register broadcast actions as protected: `/system/framework`, `/system/priv-app`, `/vendor/priv-app`, `/odm/priv-app`, & `/product/priv-app`. Except for the `/system/framework` directory, each directory ends with `priv-app`. In Android Oreo, only the `system` directory separated pre-installed apps into corresponding `app` and `priv-app` directories, although both could successfully register protected broadcasts. With Android Pie, the `app/priv-app` division scheme was also applied to the following directories: `/vendor`, `/odm`, & `/product`. Notably, the system only scans the the `/oem/app` directory for apps and not the `/oem/priv-app` directory.

On Android Pie, only pre-installed apps that reside within the `/system/framework`, `/system/priv-app`, `/vendor/priv-app`, `/odm/priv-app`, & `/product/priv-app` directories *can* register protected broadcasts, whereas pre-installed apps that reside in the following directories *cannot* register protected broadcasts: `/system/app`, `/vendor/app`, `/odm/app`, `/oem/app`, `/product/app`, `/vendor/overlay`, & `/product/overlay`. Effectively, starting with Android 9, a pre-installed app must have a valid APK path where it is contained in the `/system/framework` directory or a white-listed `priv-app` directory to successfully register a broadcast action. Android 10 is the same as Android Pie with regard to the apps in directories that it lets successfully register

[4] https://source.android.com/security/bulletin/2020-09-01
[5] https://nvd.nist.gov/vuln/detail/CVE-2020-0391
[6] https://android.googlesource.com/platform/frameworks/base/+/860fd4b6a2a4fe5d681bc07f2567fdc84f0d1580

protected broadcasts, except that it also allows apps in the `/product_services/priv-app` directory to register protected broadcasts and additionally prevents apps in the `/product_services/overlay`, `/odm/overlay`, & `/oem/overlay` directories from registering protected broadcasts.

We examined the `master` branch of AOSP code and the vulnerability was present as of May 8, 2020. In the `master` branch, only pre-installed apps in the following directories can successfully register protected broadcasts: `/system/priv-app`, `/vendor/priv-app`, `/odm/priv-app`, `/product/priv-app`, & `/system_ext/priv-app`. Apps in the following directories will fail to register protected broadcasts: `/system/app`, `/vendor/app`, `/oem/app`, `/product/app`, `/system_ext/app`, & all `overlay` directories (e.g., `/*/overlay`). Google recently released the Android 11 Developer Preview 3 and the behavior is the same as the `master` branch of AOSP for apps registering protected broadcasts. Table 1 provides summary information for various Android versions with regard to which pre-installed apps can and cannot register protected broadcasts. This table covers pre-installed apps and ignores third-party apps since they cannot register protected broadcasts.

Table 1. Android versions and how APK location affects the protected broadcast primitive in pre-installed apps.

Android Version	**Directories Where Apps *Can* Use the Protected Broadcast Primitive**	**Directories Where Apps *Cannot* Use the Protect Broadcast Primitive**
AOSP `master` branch (*as of May 8, 2020*)	`/system/framework`, `/system/priv-app`, `/vendor/priv-app`, `/odm/priv-app`, `/product/priv-app`, & `/system_ext/priv-app`	`/system/app`, `/vendor/app`, `/oem/app`, `/odm/app`, `/product/app`, `/system_ext/app`, & all `overlay` directories (i.e., `/*/overlay`)
	https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/pm/PackageManagerService.java - line 11599 has the logic to drop any protected broadcasts if the app was not scanned with the `SCAN_AS_PRIVILEGED` flag. Line 2747 starts setting the scan flags for the pre-installed app directories.	
11 (R) Developer Preview 3	`/system/framework`, `/system/priv-app`, `/vendor/priv-app`, `/odm/priv-app`, `/product/priv-app`, & `/system_ext/priv-app`	`/system/app`, `/vendor/app`, `/oem/app`, `/odm/app`, `/product/app`, `/system_ext/app`, & all `overlay` directories (i.e., `/*/overlay`)
	https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-r-preview-3/services/core/java/com/android/server/pm/PackageManagerService.java - line 11599 has the logic to drop any protected broadcasts if the app was not scanned with the `SCAN_AS_PRIVILEGED` flag. Line 2747 starts setting the scan flags for the pre-installed app directories.	
10	`/system/framework`, `/system/priv-app`, `/vendor/priv-app`, `/odm/priv-app`, `/product/priv-app`, & `/product_services/priv-app`	`/system/app`, `/vendor/app`, `/odm/app`, `/oem/app`, `/product/app`, `/product_services/app`, `/product/overlay`, `/vendor/overlay` `/product_services/overlay`, `/odm/overlay`, & `/oem/overlay`
	https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android10-release/services/core/java/com/android/server/pm/PackageManagerService.java - line 11623 has the logic to drop any protected broadcasts if the app was not scanned with the `SCAN_AS_PRIVILEGED` flag. Line 2601 starts setting the scan flags for the pre-installed app directories.	
Pie (9.0)	`/system/framework`, `/system/priv-app`, `/vendor/priv-app`, `/odm/priv-app`, & `/product/priv-app`	`/system/app`, `/vendor/app`, `/odm/app`, `/oem/app`, & `/product/app`, `/vendor/overlay`, & `/product/overlay`
	https://android.googlesource.com/platform/frameworks/base/+/refs/heads/pie-release/services/core/java/com/android/server/pm/PackageManagerService.java - line 10847 has the logic to drop any protected broadcasts if the app was not scanned with	

	the SCAN_AS_PRIVILEGED flag. Line 2600 starts setting the scan flags for the pre-installed app directories.	
Oreo (8.0)	/system/framework, /system/app, /system/priv-app, /vendor/app, /oem/app, & /vendor/overlay	N/A
	https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-8.1.0_r71/core/java/android/content/pm/PackageParser.java - line 2441 has adds protected broadcasts if the app was scanned with the PARSE_IS_SYSTEM flag. https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-8.1.0_r71/services/core/java/com/android/server/pm/PackageManagerService.java - line 2604 starts setting the parse flags for the pre-installed app directories.	

Third-party apps cannot declare protected broadcasts and have them be successfully recognized and protected by the system. As mentioned previously, the APK location of an app is considered when determining if the system should register a protected broadcast that an app declares in its manifest file. In addition, any pre-installed app that has been updated outside of a regular firmware update will have its corresponding update APK file reside in the /data/app directory. Since this app is an updated pre-installed app, it can still successfully register protected broadcasts if its initial APK location permits the successful registration of protected broadcasts. Table 1 provides links to the classes in AOSP that control which app directories containing pre-installed apps can and cannot successfully protect broadcasts. On most of the Android versions in Table 1, the PackageManagerService class parses known pre-installed app directories and contains the logic for whether their protected broadcasts should indeed be protected. In Android Oreo, the PackageParser class also contains logic for determining which protected broadcasts should be registered with the system.

3. Sample of Impacted Android Devices

Each protected-broadcast declared in an app wherein the broadcast action will not be protected can manifest as a vulnerability that negatively impacts the end-user. Android has hundreds of vendors that have the potential to introduce (un)protected broadcast vulnerabilities into their builds. While it is infeasible for us to collect and analyze all Android Pie and Android 10 builds for every vendor/model combination, we provide a sample of 89 different firmware showing the breadth of the issues in Table 2. For all firmware for each vendor, we provide the total number of pre-installed apps that declare at least one protected broadcast which will not actually be protected by the system and also the total number broadcast actions that are used with the protected broadcast primitive, but will not offer any protection. We determined the aggregate number for these two metrics for each vendor and also provided the unique number of instances for apps and broadcast actions. We consider an app to be unique based on the combination of its package name, version code, and version name.

Table 2. A sample of Android Pie and Android 10 firmware images and their exposure to the (un)protected broadcast vulnerability, providing per-vendor counts of affected apps and broadcast actions.

Vendor	**# Firmware**	**# Apps Declaring (Un)protected Broadcasts**	**# (Un)protected Broadcasts Actions Declared**	**# Unique Apps Declaring (Un)protected Broadcasts**	**# Unique (Un)protected Broadcast Actions Declared**
Google	53	295	606	8	17
Samsung	14	73	76	12	6
Xiaomi	14	143	519	30	59
Nvidia	3	27	99	9	33

Oppo	2	8	32	8	32
Asus	1	7	23	7	23
ZTE	1	3	15	3	15
Fairphone	1	6	31	6	31
Total	**89**	**562**	**1,401**	**83**	**216**

In this sample of 89 firmware, every single firmware contained at least one app that uses the protected-broadcast primitive and due to the file system location of the declaring APK file, the broadcast action will not be protected by the system and be open to spoofing by *any* app on the device. This false sense of security by trusting the provenance of broadcast intent messages with action strings that have been declared to be protected and are not due to the declaring app's APK path, results in a vulnerability that may or may not be exploitable. In the sample, there were 1,401 broadcast actions (216 unique) that were declared to be protected, although they can be spoofed by non-system users (e.g., third-party apps). There were a total of 562 apps (83 unique) that declared at least one protected broadcast action that will not be protected by the system.

4. Specific Examples

To demonstrate the impact of the (un)protected broadcast vulnerability, we provide the following example vulnerabilities that manifest due to third-party apps being able to spoof broadcast intent messages that pre-installed apps have declared as protected, but no protection is granted by the system due to the APK path. There are likely additional privilege escalation vulnerabilities residing in pre-installed apps where developers make a false assumption that certain declared broadcast actions cannot be sent by non-system users on various Android Pie builds and higher.

4.1. Various Android Vendors - `com.qualcomm.qti.perfdump`

An app with a package name of `com.qualcomm.qti.perfdump` app comes pre-installed on multiple Android devices and Android vendors. When this app is pre-installed on an Android device that uses Android version 9 or higher and its APK location is not in a white-listed `priv-app` directory, it exposes **arbitrary command execution as the `system` user** to any process on the device, including third-party apps. In addition, the app can be made to write various sensitive data (e.g., system logs, `bugreport`, screenshot, etc.) to external storage. We responsibly disclosed this vulnerability to Qualcomm, and they assigned it a CVE-ID of CVE-2020-11164. The description of the vulnerability provided by Qualcomm states "Third-party app may also call the broadcasts in Perfdump and cause privilege escalation issue due to improper access control." Qualcomm's provided the list of impacted chipsets as "Agatti, APQ8096AU, APQ8098, Bitra, Kamorta, MSM8909W, MSM8917, MSM8940, Nicobar, QCA6390, QCM2150, QCS605, Rennell, SA6155P, SA8155P, Saipan, SDA660, SDM429W, SDM450, SDM630, SDM636, SDM660, SDM670, SDM710, SM6150, SM7150, SM8150, SM8250, SXR1130, SXR2130."[7] This shows the true depth of devices that contain the aforementioned chipsets and is also posted by Qualcomm in their October 2020 security bulletin rating it as a high privilege escalation vulnerability.[8] Although Qualcomm provided the chipsets affected, we have provided a sampling of affected

[7] https://nvd.nist.gov/vuln/detail/CVE-2020-11164

[8] https://www.qualcomm.com/company/product-security/bulletins/october-2020-security-bulletin#_cve-2020-11164

Android devices in Table 3 that contain a pre-installed `com.qualcomm.qti.perfdump` app version that is vulnerable to command execution as the `system` user and leaking of sensitive data to external storage.

Table 3. Devices that contain a vulnerable version of the pre-installed `com.qualcomm.qti.perfdump` app.

Vendor	Model	Product Name	Android Version	App Version Code	App Version Name
Sony	Xperia 1	`802SO`	9	8	3.0.1
Nokia	7 Plus	`B2N_sprout`	9	7	2.1.1
Fairphone	Fairphone 3	`FP3`	9	8	3.0.1
Meizu	Note 9	`meizunote9`	9	7	2.1.1
Meizu	16Xs	`meizu16Xs`	9	8	3.0.1
Xiaomi	Poco F1	`beryllium`	9	7	2.1.1
Xiaomi	Mi 9	`cepheus`	9	7	2.1.1
Xiaomi	Mi 8	`dipper`	9	7	2.1.1
Xiaomi	Mi 8 Pro	`equuleus`	9	7	2.1.1
Xiaomi	Mi Max 3	`nitrogen`	9	7	2.1.1
Xiaomi	Mi Mix 3	`perseus`	9	7	2.1.1
Xiaomi	Mi 8 Lite	`platina`	9	7	2.1.1
Xiaomi	Mi A2 Lite	`daisy`	9	7	2.1.1
Xiaomi	Mi A2	`jasmine`	9	7	2.1.1
Xiaomi	Redmi Note 7	`lavender`	9	7	2.1.1
Xiaomi	Redmi 7	`onc_eea`	9	7	2.1.1
Xiaomi	Redmi Note 8T	`willow_eea`	9	8	3.0.1

We verified the command execution vulnerability on a Xiaomi Redmi Note 8T Android device which contains `com.qualcomm.qti.perfdump` (versionCode=8, versionName=3.0.1) as a pre-installed app with an APK path of `/system/app/Perfdump/Perfdump.apk`. The APK path prevents the `com.qualcomm.qti.perfdump` app from successfully protecting the broadcast actions it *attempts* to protect. The `com.qualcomm.qti.perfdump` app is signed with the platform key and executes with the `system` UID (e.g., `1000`) on the Xiaomi Redmi Note 8T device (`xiaomi/willow_eea/willow:9/PKQ1.190616.001/V10.3.0.6.PCXEUOR:user/release-keys`), providing significant privilege when executing commands.

This app declares seven different protected broadcasts in its `AndroidManifest.xml` file. Notably, the app declares an action string named `android.perfdump.action.EXT_EXEC_SHELL` as a protected broadcast in its manifest file. Also in its manifest, it statically registers the `com.qualcomm.qti.perfdump.StaticReceiver` broadcast receiver to receive broadcast intents with the `android.perfdump.action.EXT_EXEC_SHELL` action string. When the `StaticReceiver` broadcast receiver receives a broadcast intent with an action string of `android.perfdump.action.EXT_EXEC_SHELL`, it forwards the intent to a service app component named `com.qualcomm.qti.perfdump.ExtRequestService`. The `ExtRequestService` extracts a string extra from the received intent named `shellCommand`. The service ensures that command is not empty, and starts a thread to execute the `sh -c <value of shellCommand>` command using the `java.lang.Runtime.exec(String[])` Application Programming Interface (API) method call. Effectively, the service application component named `ExtRequestService` executes an externally-controlled command in a non-interactive shell, giving the attacker greater capability than executing a command without the shell environment. In addition, the command is completely controlled by the attacker since the whole command originates from the attacker, except for the `sh -c` portion that conveniently provides the non-interactive shell. Listing 4 provides the source code to execute arbitrary commands as the `system` user via the `com.qualcomm.qti.perfdump` app on impacted devices.

```
Intent intent = new Intent("android.perfdump.action.EXT_EXEC_SHELL");
intent.setClassName("com.qualcomm.qti.perfdump",
"com.qualcomm.qti.perfdump.StaticReceiver");
intent.putExtra("callerPackageName", "com.test");
intent.putExtra("shellCommand", <command_to_execute>);
sendBroadcast(intent);
```

Listing 4. Source code snippet to execute commands as system UID via the `com.qualcomm.qti.perfdump` app.

If there is any output on the standard error stream from the executed command, the app writes the last line from the standard error stream to the system log using a log tag of `PERFDUMP.EXT` and also sends it in a non-permission protected broadcast intent with an action string of `android.perfdump.action.EXT_FEEDBACK` to the package name that was provided in the `callerPackageName` string extra in the broadcast intent that it receives.

The `com.qualcomm.qti.perfdump` app also tries to register the `android.perfdump.action.EXT_START_TRACE` and `android.perfdump.action.EXT_DUMP_TRACE` actions as protected broadcasts, although they will not be protected due to the file system location of the APK file. The `com.qualcomm.qti.perfdump.StaticReceiver` broadcast receiver statically registers to receive broadcast intent messages that have action strings named `android.perfdump.action.EXT_START_TRACE` and `android.perfdump.action.EXT_DUMP_TRACE`. These two broadcast actions can be sent by third-party apps which causes the `com.qualcomm.qti.perfdump` app to write various system logs and debugging information to external storage on the device. Notably, the app dumps the following to external storage: `logcat` log, `dumpsys` output, screenshot, and a `bugreport`. These system logs and debugging information are not directly available to third-party apps since they tend to contain sensitive information. The sensitive data that the app dumps to external storage is controlled by various boolean extras that are sent in the broadcast intent with an action string of `android.perfdump.action.EXT_START_TRACE`. The `dumpsys` output and especially the `bugreport` contain extensive information about the state of the system. An external app selects a directory on the file system where `com.qualcomm.qti.perfdump` app will dump the sensitive data using string extras in the intent. We caused the `com.qualcomm.qti.perfdump` app to dump the sensitive data to a directory on external storage. On the Xiaomi Redmi Note 8T device, SELinux blocked the `com.qualcomm.qti.perfdump` app from writing the sensitive data directly to a third-party app's private directory. This can be overcome by having the system UID read the file on external storage in chunks and then send the constituent data to the attack app via a dynamically registered broadcast receiver.[9] Whether or not the com.qualcomm.qti.perfdump app can write to a third-party app's private directory depends on the specific SELinux policies of the device. A third-party app can use the (un)protected broadcasts of the com.qualcomm.qti.perfdump app to obtain sensitive information about the user and also the state of the system.

4.2 Various Android Vendors - com.qualcomm.qti.qmmi

We verified the a vulnerability on an a number of Android devices which contain com.qualcomm.qti.qmmi (versionCode=400, versionName=4.0) as a pre-installed app with an APK path of /system/app/Qmmi/Qmmi.apk. The APK path prevents the com.qualcomm.qti.qmmi app from successfully protecting the broadcast actions it attempts to protect. This app is signed with the platform key and executes with the system UID (e.g., 1000), affording the app significant capabilities. These vulnerabilities generally do not apply if the com.qualcomm.qti.qmmi app is in a priv-app directory (e.g., /system/priv-app/Qmmi/Qmmi.apk), although we have only seen the com.qualcomm.qti.qmmi app have a path of /system/app/Qmmi/Qmmi.apk.

The com.qualcomm.qti.qmmi app declares six different protected broadcasts in its AndroidManifest.xml file. Notably, the app declares an action string named qualcomm.qti.qmmi.DIAG_START_TESTCAST as a protected broadcast in its manifest file. The com.qualcomm.qti.qmmi app dynamically registers an anonymous class (i.e., com.qualcomm.qti.qmmi.framework.MainActivity$1) to receive broadcast intents with the qualcomm.qti.qmmi.DIAG_START_TESTCAST action string. When the MainActivity$1 broadcast receiver receives a broadcast intent with an action string of qualcomm.qti.qmmi.DIAG_START_TESTCAST, it will check for a string extra in the received intent named case_name and its corresponding value is the name of the test case that should be run. The com.qualcomm.qti.qmmi app has a list of valid test cases that it populates from embedded XML files. This vulnerability focuses on the test case named SYSTEM_INFO.: SYSTEM_INFO. Each test case (e.g., WIFI, BLUETOOTH, and NFC) generally starts either a corresponding activity app component and service app component.

This vulnerability allows a third-party app with no permissions to obtain the following unique device identifiers: IMEI1, IMEI2 (if present), Wi-Fi MAC address, Bluetooth address, and serial number. Qualcomm assigned this vulnerability a CVE ID of CVE-2021-1929, although it is still reserved and not populated at the time of writing. To exploit the vulnerability, the third party-app first starts the com.qualcomm.qti.qmmi.framework.MainActivity activity component using an intent. The com.qualcomm.qti.qmmi.framework.MainActivity component will dynamically register an anonymous broadcast receiver, named com.qualcomm.qti.qmmi.framework.MainActivity$1, to receive intents with an action string of qualcomm.qti.qmmi.DIAG_START_TESTCAST. Although the com.qualcomm.qti.qmmi app declares qualcomm.qti.qmmi.DIAG_START_TESTCAST as a protected broadcast in its AndroidManifest.xml file, the system will not grant it protection since the app does not reside within a priv-app directory (e.g., /system/priv-app). Therefore, any app, including third-party apps, can send a broadcast intent with any action string the com.qualcomm.qti.qmmi app has declared as protected in its manifest file. Then the third-party app sends a broadcast intent with an action string of qualcomm.qti.qmmi.DIAG_START_TESTCAST and a string extra named case_name with a value of SYSTEM_INFO. When the this broadcast intent is received, the com.qualcomm.qti.qmmi app will start the com.qualcomm.qti.qmmi.testcase.SystemInfo.SystemInfoService app component to gather system

[9] Section 4.1.1 of this DEF CON 26 paper contain details about transferring data via implicit broadcast messages: https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20presentations/Ryan%20Johnson%20and%20Angelos%20Stavrou%20-%20Updated/DEFCON-26-Johnson-and-Stavrou-Vulnerable-Out-of-the-Box-An-Eval-of-Android-Carrier-Devices-WP-Updated.pdf

information and also start the com.qualcomm.qti.qmmi.testcase.SystemInfo.SystemInfoActivity app component to display the system information it receives to the user. The SystemInfoService app component obtains the various unique device identifiers (IMEI1, IMEI2 (if present), Wi-Fi MAC address, Bluetooth address, and serial number) and after these device identifiers have been gathered, they will be put into an intent with a corresponding intent string intent name of msg. It will then send the intent with an action name of qualcomm.qti.qmmi.UPDATE_MESSAGE as an ordered broadcast message. This broadcast intent is an implicit intent since there is not a specific app package name or destination component that is declared as the recipient of the intent. In addition, the com.qualcomm.qti.qmmi app does not require that the receiving process possess a permission. Therefore, any app that registers for the qualcomm.qti.qmmi.UPDATE_MESSAGE action can receive the implicit broadcast intent that the com.qualcomm.qti.qmmi app sends that contains the unique devices identifiers. Listing 5 provides the source code to programmatically obtain the unique device identifiers from the com.qualcomm.qti.qmmi app. The source code will simply write the unique device identifiers to the logcat log with a log tag of msg. The actual log messages obtained from a device using this code are highlighted in the listing.

```
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction("qualcomm.qti.qmmi.UPDATE_MESSAGE");
registerReceiver(new BroadcastReceiver() {
   @Override
   public void onReceive(Context context, Intent intent) {
       if (intent == null || !intent.hasExtra("msg"))
           return;
       if ("qualcomm.qti.qmmi.UPDATE_MESSAGE".equals(intent.getAction())) {
           String msg = intent.getStringExtra("msg");
           String[] msg_strs = msg.split("\\n");
           for (String msg_str : msg_strs) {
               Log.d("msg", msg_str);
       // when the broadcast intent is received, it will contain the following data
       // D msg     : Android Version:9
       // D msg     : Modem:Q_V1_P14,Q_V1_P14
       // D msg     : Serial:9655cfac
       // D msg     : IMEI1:863112046217716
       // D msg     : IMEI2:863112046217708
       // D msg     : BT Address:18:D0:C5:E0:63:A8
       // D msg     : WIFI MAC:18:d0:c5:e0:63:a9
       // D msg     : Diag support:YES
           }
       }
   }
}, intentFilter);

Intent intent = new Intent();
intent.setClassName("com.qualcomm.qti.qmmi",
"com.qualcomm.qti.qmmi.framework.MainActivity");
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK
Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
startActivity(intent);

try {
   Thread.sleep(3500);
} catch (InterruptedException e) {
   e.printStackTrace();
}

Intent broadcast_intent = new Intent();
broadcast_intent.setAction("qualcomm.qti.qmmi.DIAG_START_TESTCAST");
broadcast_intent.putExtra("case_name", "SYSTEM_INFO");
```

```
sendBroadcast(broadcast_intent);

try {
   Thread.sleep(7000);
} catch (InterruptedException e) {
   e.printStackTrace();
}

Intent home_screen_intent = new Intent("android.intent.action.MAIN");
home_screen_intent.addCategory("android.intent.category.HOME");
home_screen_intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
startActivity(home_screen_intent);
```

Listing 5. Java source code to obtain the unique device identifiers.

In addition to sending the unique device identifiers using an implicit intent, these unique device identifier values are written to the logcat log by the `com.qualcomm.qti.qmmi` app. Listing 6 provides the logcat messages written by the `com.qualcomm.qti.qmmi` app with the relevant portions of the message highlighted.

```
I Qmmi    : [ResultParser.java] [saveResultToFile] writeResultFile, TestCase , data size:8
I Qmmi    : [ResultParser.java] [saveResultToFile] write ressult file, data
,key:imei2,param,value:863112046217708
I Qmmi    : [ResultParser.java] [saveResultToFile] write ressult file, data
,key:bt_address,param,value:18:D0:C5:E0:63:A8
I Qmmi    : [ResultParser.java] [saveResultToFile] write ressult file, data
,key:imei1,param,value:863112046217716
I Qmmi    : [ResultParser.java] [saveResultToFile] write ressult file, data
,key:serial,param,value:9655cfac
I Qmmi    : [ResultParser.java] [saveResultToFile] write ressult file, data
,key:modem,param,value:Q_V1_P14,Q_V1_P14
I Qmmi    : [ResultParser.java] [saveResultToFile] write ressult file, data
,key:wifi_mac,param,value:18:d0:c5:e0:63:a9
```

Listing 6. Unique device identifiers leaked to the logcat log.

Only pre-installed apps and apps signed with the framework key can obtain the system-wide logcat log. Therefore, third-party apps cannot obtain the system-wide logcat log directly. We have seen numerous instances where the logcat log is leaked by a pre-installed app, so that it becomes accessible to a third-party app. So writing to the logcat log does not guarantee that third-party apps will not be able to indirectly obtain it.

4.3 Pixel Devices - `com.android.service.ims.presence`

Various Google Pixel devices contain a pre-installed app that facilitates Rich Communication Services (RCS) with a package name of `com.android.service.ims.presence`. As part of this app's functionality, it maintains a database that "mirrors" the device's official contacts provider. On the Pixel devices, this APK has a file system location of `/system/app/PresencePolling/PresencePolling.apk`. This app uses the protected-broadcast primitive in its `AndroidManifest.xml` file to try to prevent non-system processes from broadcasting intents containing four different action strings. Due to the file system location of the app, the action strings that the app declares to protect will not be registered by the system as protected broadcasts, allowing any process on the device to send broadcast messages with the action strings that the app *attempts* to register as protected.

Of the four action strings the app attempts to register as protected broadcasts, one is named `android.provider.rcs.eab.EAB_DATABASE_RESET`. This action is sent by the app when it upgrades or downgrades an internal database named `rcseab.db`. The `com.android.service.ims.presence.EABService` service app component creates a broadcast receiver implementation and dynamically registers it for the `android.provider.rcs.eab.EAB_DATABASE_RESET` action, among a few others. When the app's broadcast receiver receives an intent with an action string of `android.provider.rcs.eab.EAB_DATABASE_RESET`, it will initiate a synchronization of the `EABPresence` table in the `rcseab.db` file with the device's official contact provider. This causes the `com.android.service.ims.presence` app to query for all of the contact info from the official contact provider and write it to the `EABPresence` table in the `rcseab.db` file, independent of whether the info for a contact already exists in the `EABPresence` table or not. A third-party app with no permissions can send a broadcast intent with an action string of `android.provider.rcs.eab.EAB_DATABASE_RESET` which will be received by the `com.android.service.ims.presence` app. Listing 7 provides a snippet of the `dumpsys` command output showing that the `com.android.service.ims.presence` app registration for the `android.provider.rcs.eab.EAB_DATABASE_RESET` action string using a broadcast receiver. The `com.android.service.ims.presence` app uses a process name of `com.android.ims.rcsservice` due a chosen configuration in its manifest file (i.e. `android:process`). This broadcast receiver does not require that the sender of broadcast intents containing these actions possess any specific permission.

```
  * ReceiverList{6faeef 3157 com.android.ims.rcsservice/1001/u0 remote:ad40fce}
    app=3157:com.android.ims.rcsservice/1001 pid=3157 uid=1001 user=0
    Filter #0: BroadcastFilter{2b47dfc}
      Action: "android.provider.Contacts.DATABASE_CREATED"
      Action: "android.intent.action.TIME_SET"
      Action: "android.intent.action.TIMEZONE_CHANGED"
      Action: "android.provider.rcs.eab.EAB_DATABASE_RESET"
```

Listing 7. Partial `dumpsys` output showing a broadcast receiver in the `com.android.service.ims.presence` app having registered to receive the `android.provider.rcs.eab.EAB_DATABASE_RESET` action string.

The `com.android.service.ims.presence` app registers for the four actions strings shown in the listing. If the `android.provider.rcs.eab.EAB_DATABASE_RESET` action does not immediately show up in the `dumpsys` output, then insert a Subscriber Identity Module (SIM) and accept the terms of using RCS when presented with a dialog in the standard Messages app. Since the app tries to register the `android.provider.rcs.eab.EAB_DATABASE_RESET` action as a protected action and since the app has an APK file system location (i.e., `/system/app/PresencePolling/PresencePolling.apk`) that is not in an authorized `priv-app` directory, the protected broadcast actions will not be protected, and non-system processes, including third-party apps will be able to send broadcast intents with these actions. Therefore, a third-party app residing on the device, can send a broadcast intent with an action string of `android.provider.rcs.eab.EAB_DATABASE_RESET` which the `com.android.service.ims.presence` app *attempted* to register as a protected broadcast. Listing 8 shows partial output of the `dumpsys` command displaying a third-party app with no permissions spoofing a broadcast intent with an action of `android.provider.rcs.eab.EAB_DATABASE_RESET` that the `com.android.service.ims.presence` app *attempted* to register as protected broadcast but *failed* to do so.

```
  Historical Broadcast background #0:
    BroadcastRecord{21e6a6 u0 android.provider.rcs.eab.EAB_DATABASE_RESET} to user 0
    Intent { act=android.provider.rcs.eab.EAB_DATABASE_RESET flg=0x10 }
    caller=com.kryptowire.thirdpartyapp 25608:com.kryptowire.thirdpartyapp/u0a148
pid=25608 uid=10148
    enqueueClockTime=2020-04-29 14:01:56 dispatchClockTime=2020-04-29 14:01:56
    dispatchTime=-15ms (0 since enq) finishTime=-15ms (0 since disp)
    Deliver #0: BroadcastFilter{2b47dfc u0 ReceiverList{6faeef 3157
com.android.ims.rcsservice/1001/u0 remote:ad40fce}}
```

Listing 8. Partial `dumpsys` output showing that the `com.android.service.ims.presence` app received a broadcast intent with the `android.provider.rcs.eab.EAB_DATABASE_RESET` action string from a third-party app named `com.kryptowire.thirdpartyapp`.

When the spoofed broadcast intent with an action string of android.provider.rcs.eab.EAB_DATABASE_RESET is received by the com.android.service.ims.presence app, it will obtain all of the user's contacts from the default contact provider and copy them to the EABPresence table in the rcseab.db file. There is no check to see if contact entries already exist prior to copying them, so that the app adds all of the user's contacts to its EABPresence table each time is receives a broadcast intent with the android.provider.rcs.eab.EAB_DATABASE_RESET action string. Therefore, the EABPresence table, and the underlying rcseab.db file, increases in size by the number of user contacts in the default contact provider each time it receives an intent with an action string that it intended to protect. A third-party app can repeatedly send the android.provider.rcs.eab.EAB_DATABASE_RESET action string in a broadcast intent and continually increase the size of the EABPresence table in the rcseab.db file. When a third-party app persists in sending this broadcast action intent that should be protected, it can increase the size of the rcseab.db file and potentially cause a Denial of Service (DoS) attack on an pre-installed app that executes as the radio user with a UID of 1001. The usage of the protected broadcast primitive is not limited to the com.android.service.ims.presence app on Pixel devices. Table 4 displays the apps on an Android Pixel 4 build (google/flame/flame:10/QQ2A.200405.005/6254899:user/release-keys) running Android 10 that declared a protected broadcast which will not be protected based on the file path of the declaring app. All of the apps in Table 4 have a version code of 29 and a version name of 10.

Table 4. Apps on an Android 10 Pixel 4 build that declare a protected broadcast that will not be protected.

Package Name	Protected Broadcasts Declared	App Path on Device
com.qualcomm.qti.uceShimService	4	/product/app/uceShimService/uceShimService.apk
com.google.SSRestartDetector	2	/product/app/SSRestartDetector/SSRestartDetector.apk
com.android.service.ims.presence	4	/system/app/PresencePolling/PresencePolling.apk

4.4 Fairphone 3 - com.qualcomm.qti.logkit.lite

An app with a package name of com.qualcomm.qti.logkit.lite (versionCode=3, versionName=4.00.000) comes pre-installed on the Fairphone 3 with a path of /vendor/app/qti-logkit/qti-logkit.apk. The device has a build fingerprint of Fairphone/FP3/FP3:9/8901.2.A.0096.20191001/10011803:user/release-keys. The com.qualcomm.qti.logkit.lite app declares five different protected broadcasts in its manifest file, but none of them will be protected based on the APK path. One of the (un)protected broadcasts which has an action string of com.qualcomm.qti.logkit.lite.intent.action.cAutomation.Automation can be used by third-party apps to cause the com.qualcomm.qti.logkit.lite app to leak the logcat log to external storage. The app exploiting this vulnerability must first start the com.qualcomm.qti.logkit.lite.cActivity activity app component and then they send a broadcast intent with an action string of com.qualcomm.qti.logkit.lite.intent.action.cAutomation.Automation and a string extra named Logging with a value of Start. After any arbitrary period of time has passed, the app can send another broadcast intent with the same action string and a string extra named Logging with a value of Stop to stop the logging. Finally, the app can send a broadcast intent the same action string and a string extra named Package with a value of -sdcard1 which will cause the system-wide logcat log to be leaked to external storage. The system-wide logcat log is not available to third-party apps directly, but a third-party app can use the broadcast actions that the `com.qualcomm.qti.logkit.lite` app tries to register as protected to indirectly obtain the system-wide `logcat` log after it is leaked to external storage. Listing 9 displays a source code snippet that will cause the `com.qualcomm.qti.logkit.lite` app to leak a logcat file on external storage, specifically to the `/sdcard/Android/data/com.qualcomm.qti.logkit.lite/files/logdata` directory.

```
startActivity(new Intent().setClassName("com.qualcomm.qti.logkit.lite",
"com.qualcomm.qti.logkit.lite.cActivity"));
Intent start_intent = new
Intent("com.qualcomm.qti.logkit.lite.intent.action.cAutomation.Automation");
start_intent.putExtra("Logging", "Start");
sendBroadcast(start_intent);
// sleep ten seconds
Thread.sleep(10000);
Intent stop_intent = new
Intent("com.qualcomm.qti.logkit.lite.intent.action.cAutomation.Automation");
stop_intent.putExtra("Logging", "Stop");
sendBroadcast(stop_intent);
Intent dump_intent = new
Intent("com.qualcomm.qti.logkit.lite.intent.action.cAutomation.Automation");
dump_intent.putExtra("Package", "-sdcard0");
sendBroadcast(dump_intent);
```

Listing 9. Source code snippet to dump logs to external storage.

5. Conclusion

This writeup outlined a vulnerability that we discovered in Android versions 9, 10, and 11 Developer Preview 3 that allowed unauthorized apps to spoof sensitive messages and send them to unsuspecting apps as if they were sent by a legitimate source. On unpatched devices, the Android system silently ignores protection requests of broadcast actions declared by a pre-installed app if the app is not pre-installed at certain paths on the filesystem. We found numerous vulnerable instances in the wild in which this vulnerability could enable third party apps to escalate privileges and perform sensitive functionalities by delegating requests to vulnerable pre-installed apps (a confused-deputy attack). We disclosed the vulnerability to Google and patches have rolled out for impacted versions. Providing explicit feedback to the developers or a runtime warning for pre-installed apps may be helpful in catching this and similar scenarios in the future before damage is done to end users.