Why Software DoS is Hard to Fix: Denying Access in Embedded Android Platforms

Ryan Johnson[✉,1,2], Mohamed Elsabagh[1], and Angelos Stavrou[1,2]

[1]George Mason University, Fairfax, VA 22030, USA
melsabag@gmu.edu
[2]Kryptowire, Fairfax, VA 22030, USA
{rjohnson, astavrou}@kryptowire.com

Abstract. A new class of software Denial of Service (DoS) attacks against Android platforms was recently discovered, where the attacks can force the victim device unresponsive, target and terminate other applications on the device, and continuously soft reboot the device [26]. After Google was informed of these DoS attacks, their attempt to resolve the problem did not adequately address the fundamental underlying attack principles. In this paper, we show that engineering software DoS defenses is challenging, especially for embedded and resource-constrained devices. To support our findings, we detail a revised DoS attack strategy for the latest version of Android. For our experimental evaluation, we demonstrate that the new class of DoS attacks are even more damaging to embedded Android devices. As part of our proof-of-concept attacks, we were able to render the Sony Bravia XBR-43X830C Android TV and the Amazon Fire TV Stick 1st generation devices permanently unusable. In addition, other devices, including the Moto 360 1st generation smartwatch, required flashing firmware images, whereas the Nvidia Shield Android TV and the Amazon Fire 7″ Tablet required a factory reset to recover. Our attack is applicable to most Android devices and requires manual intervention to attempt to recover the device. The proposed attack strategy is more debilitating to devices that do not provide means for the end-user to easily access safe mode, recovery mode, or the ability flash firmware images. To mitigate the attack, we created an open-source defense application that has a 100% prevention rate after a single soft reboot of the device while incurring less than 1.6% performance overhead.

Keywords: Android; DoS Attack; DoS Defense; Mobile Security.

1 Introduction

The Android Operating System (OS) is becoming popular and pervasive to embedded platforms such as mini PCs, streaming media players, smart TVs, smartwatches, and infotainment systems. Despite the fact that most of the underlying Android framework remains the same among these devices, a common vulnerability may affect each platform differently. This is due to the devices having

different form factors, hardware buttons, safe mode availability, and access to the recovery and fastboot modes. For instance, smartphones, the most mature of the Android platforms, are the best-equipped to deal with malicious applications since they generally provide both easy access to safe mode and recovery mode from a powered-off state by holding a combination of hardware buttons during boot. Some of the less mature or resource-constrained devices may lack or not provide easy access to these capabilities, which increases their exposure to Denial of Service (DoS) attacks.

Designing adequate defenses to software DoS attacks is difficult: in most cases, the resource under attack is shared and thus a trade-off between preventing the attack and allowing legitimate use of the resource is required in practice. If the attack countermeasure is not restrictive enough, it will enable a malicious actor to reduce the availability of the resource. On the other hand, if the attack countermeasure is too restrictive, it will limit legitimate usage of the resource. In the context of the DoS attack presented in this paper, the resource being attacked is availability of the device itself and, by extension, all of its constituent resources. Contrary to the software DoS, preventing DoS and Distributed DoS (DDoS) for network-based attacks is a well-researched area [27, 28, 31–33] and is known to be a difficult problem. There has been less research in application-level DoS attacks, which exploit inherent software design weaknesses, especially against Android [16, 21, 25, 26].

In Android, intents are used for inter-process and intra-process communication. An intent is like a message that is sent by an app to itself or another app. An intent can contain data to be utilized by the receiving app to perform an action. Broadcast intents are sent to all apps that listen for a specific event or handle an action. Intents are a fundamental communication mechanism that are used by Android apps and can be abused since the Android OS does not put any limit on the amount or rate that intents can be directly sent from an app. Rapidly sending intents from a third-party app can result in various DoS attacks including making the target device unresponsive to the user, targeting and terminating other running apps, and forcing a soft reboot of the device. A soft reboot occurs whens the Android framework, residing in user space, crashes, but the Linux kernel continues execution. A soft reboot may appear to the user as a reboot since the Android boot animation is displayed during a soft reboot.

We informed Google of a novel class of intent-based DoS attacks on Android in September 2015, and they subsequently introduced fixes in Android to address them. We created variations of the intent-based DoS attacks that work around Google's fixes, making the attacks effective on the latest Android version. In this paper, we focus on the DoS attack to quickly and repeatedly soft reboot an Android device, which we refer to as the soft reboot cycle DoS attack, since it is the most severe of the DoS attacks. We provide results for the updated soft reboot cycle DoS attack on popular embedded Android devices. The underlying cause of the soft reboot is explained in conjunction with referencing Android Open Source Project (AOSP) Android 6 source code files. We also proposed changes to the Android framework to thwart the attacks, and we created an

open-source Android application that precludes the soft reboot cycle DoS attack from being successful. This countermeasure application can be utilized by device manufacturers without making any modification to the Android framework. Device manufacturers can utilize it as a system application in their next build or sign the application with the device platform key to make it readily deliverable to current devices.

2 Threat Model

We assume that the user side-loads the malicious application or downloads and installs it from an official or third-party application marketplace. The code to perform the soft reboot cycle DoS attack can be introduced by repackaging a popular application with malicious code. Repackaging Android applications is a popular method for distributing malware [29, 30, 34–36]. Social engineering is another possible attack vector to deliver the malicious application [10,13,18,23]. The available approaches to remove an application depend on the specific Android device. Safe mode prevents the execution of installed third-party applications. If safe mode is available on the device, the user can boot into safe mode and uninstall third-party applications. Android Debug Bridge (ADB) is a command-line tool that allows the user to issue commands from a separate computing device to an Android device or emulator. ADB comes disabled by default on most devices. The user must specifically enable ADB in the Settings app, *and* authorize the debug device that the Android device will be connected to [7]. If ADB over a USB cable is enabled, then the user can obtain a list of all installed third-party applications on the device using the `adb shell pm list packages -3` command and uninstall them using ADB.

Certain devices will allow the user to boot into recovery mode and fastboot mode from a powered-off or booting state using hardware buttons or screen touches on smartwatches. The standard Android recovery mode allows a user to perform a factory reset which wipes the data and cache partitions on the device resulting in the removal of the user's installed applications. Fastboot mode allows the user to flash firmware images to the device if the bootloader can be unlocked. The soft reboot cycle DoS attack is persistent: once the attack is triggered, the device becomes unresponsive and enters into soft reboot cycles. To summarize, if all of the following four conditions are met, the user cannot remove an app executing the soft reboot cycle DoS attack from the device:

1. No access to safe mode on the device.
2. ADB over USB is disabled prior to the attack.
3. The Android OS sends the `BOOT_COMPLETED` broadcast intent to third-party apps after the booting process completes[1].
4. There is no hardware-based method to enter a mode from a powered-off or booting state that will allow the user to perform a factory reset or flash firmware images.

[1] The only Android device that we have encountered that does not do this is the Xiaomi Mi TV Box Mini [11].

If only the first 3 conditions are fulfilled, the user is forced to perform a factory reset or flash firmware images to recover the device from the attack.

3 Attack Method

Conceptual Attack Summary A third-party Android app can soft reboot the Android OS by sending a large amount of intents rapidly. An Android app is composed of application components. An activity application component provides a Graphical User Interface (GUI) that allows the user to interact with the application. A service application component performs tasks in the background and does not present a GUI to the user. A broadcast receiver application component listens for specific events and state changes that occur within an app or the OS itself. The attack app contains the following application components: activity, service, and broadcast receiver.

The attack begins shortly after the Android OS boot process completes. The OS sends a broadcast intent, to indicate the fact that the boot process has completed, to broadcast receivers who have permission to receive it. The broadcast receiver in the attack app receives this broadcast intent and starts the service application component so it can execute in the background. The service application component then starts rapidly sending intents to start the activity application component. The intents being sent by the service contain specific flags which create an activity in its own task stack, so new activities are created even though the same activity already exists in a different task stack. Each started activity will send an intent to the service which will create more activities and the cycle repeats leading to a multiplicity the same activity being created.

The `system_server` process, an integral part of the Android framework, contains service threads that apps interact with using a client-server architecture. `system_server` creates the activity application components requested the by the service, and it also creates a socket pair to deliver the user's touch events to the app. Each activity that is created requires a single file descriptor from the `system_server` process for its end of the socket pair, although it can require two file descriptors if intents are sent rapidly since it will not be able to transfer the other socket to the app. Each process has a soft limit of file descriptors to prevent a single process from exhausting the resource. Once a process hits its soft limit for file descriptors, it cannot open or create files, pipes, or sockets.

A third-party app can create activities rapidly to force `system_server` to reach its soft limit of 1,024 file descriptors. When this occurs, `system_server` is constrained and can crash in a number of ways. The most common crash is due to `system_server` trying to create a system message dialog box indicating that the attacking app has crashed. A socket for this dialog box is required to obtain the user's input, but `system_server` will not be able to create it. This leads to an uncaught exception and results in a crash of `system_server`. This event causes the Android OS to soft reboot. The attacking app will again receive the broadcast intent that is sent out to apps indicating that the Android OS has completed the

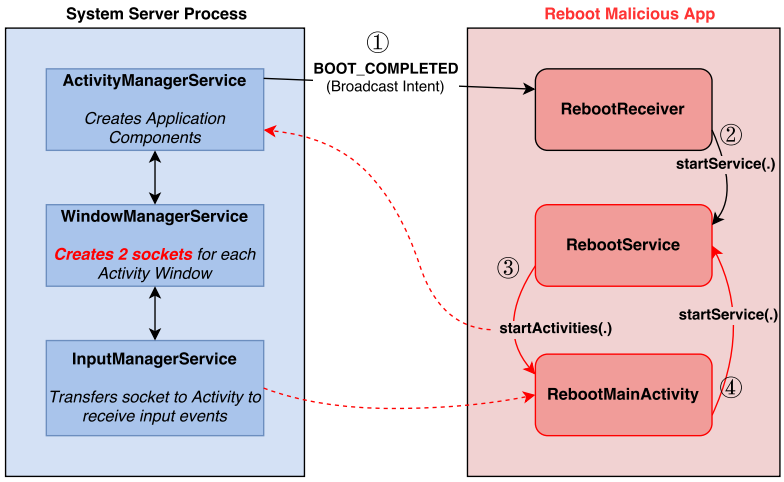

Fig. 1. Interaction between the Reboot malicious app and the `system_server` process. Dashed lines indicate indirect inter-process interactions.

boot process. The attacking app again performs the attack to make the device soft reboot and this cycle will persistently occur until the user manually takes some action to prevent it.

Prior to Android 6, a third-party app was able to make the `system_server` process attack itself and eventually crash by creating a repeating alarm to have `system_server` send an intent every millisecond. Google, in response to our vulnerability disclosure, raised the minimum recurrence interval in between alarms to 60 seconds. This partially addressed the vulnerability, although they did not add a restriction on the amount or rate for all available means that an app can send intents. An app can still send an unrestricted amount of intents directly from an application component using the inherited methods of the `android.content.ContextWrapper` class. Without rate-limiting the sending of intents for all approaches available to an app or creating a reasonable limit on the amount of activity instances an app can concurrently have, the attack will be successful.

Soft Rebooting The Device The interaction between the Reboot application, our malicious app, and `system_server` is shown in Figure 1. Certain events have been omitted from Figure 1 for clarity, such as the fact that the `com.android.server.am.ActivityManagerService` class creates all application components used by the attack app. In addition, only certain services within `system_server` are displayed. The Reboot application has a broadcast receiver application component (i.e., RebootReceiver in Figure 1) to receive the `BOOT_COMPLETED` broadcast intent sent from `system_server`, so that the application can begin execution shortly after the Android OS completes the boot process (displayed as arrow 1 in Figure 1). The app also listens for the `android.hardware.usb.action.USB_STATE` broadcast

intent which is part of AOSP and does *not* require a permission. On all the devices we tested, this broadcast intent can be received prior to the `BOOT_COMPLETED` broadcast intent. Listing 1.1 shows how RebootReceiver should be declared in the app's `AndroidManifest.xml` file. It is important to ensure that the `android:priority` attribute be set to the maximum value (i.e., 999) in the `intent-filter` for the `BOOT_COMPLETED` action. Upon receiving this intent, RebootReceiver sends an intent to start the RebootService (displayed as arrow 2 in Figure 1). RebootService will, first, create a thread to perform the attack, then return the `START_STICKY` constant in its `onStartCommand` method.

The thread that launches the attack will send a large number of intents to rapidly create numerous instances of activity application components (i.e, RebootMainActivity in Figure 1) that are internal to the attacking app. The attack requires that the intents use the following two flags: `FLAG_ACTIVITY_MULTIPLE_TASK` and `FLAG_ACTIVITY_NEW_TASK`. These intent flags, when used together, create a new task stack containing a single activity even when a matching activity already exists within the attack application. The default behavior, without using these intent flags, is to push a new activity on top of the current task stack. Newly created instances of RebootMainActivity will attempt to start the RebootService in its `onCreate` method. The RebootService has already been created and is running, so it will just result in the execution of its `onStartCommand` method which will result in the creation of more instances of RebootMainActivity. This essentially creates an cycle of the two application components calling each other (displayed as arrows 3 and 4 in Figure 1). The attack creates numerous instances of task stacks containing only a single activity. Each task stack will require `system_server` to allocate 1 to 2 file descriptors depending on the rate of the attack. The attack causes `system_server` to exhaust its file descriptors. As a result of this condition, `system_server` generally encounters an uncaught exception causing its termination. Alternatively, the watchdog daemon process can also kill `system_server` if it perceives a deadlock.

Disabling Wireless Communication Methods The attack can be made more aggressive by having the attack app programmatically disable the Bluetooth and Wi-Fi communication methods on the device. The `android.bluetooth.BluetoothAdapter.disable()` Android Application Programming Interface (API) call requires the `BLUETOOTH` and the `BLUETOOTH_ADMIN` permissions. This API call disables Bluetooth on the device so that any paired devices can no longer inter-

```
<receiver android:name="RebootReceiver">
  <intent-filter android:priority="999">
    <action android:name="android.intent.action.BOOT_COMPLETED" />
  </intent-filter>
</receiver>
```

Listing 1.1. Declaration of the `RebootReceiver` in the app's manifest.

act with the device. This will also preclude ADB over Bluetooth to the device for Android Wear devices. The `android.net.wifi.WifiManager.setWifiEnabled(boolean)` API call requires the `ACCESS_WIFI_STATE` and `CHANGE_WIFI_STATE` permissions. This API call can disable Wi-Fi so that other devices on the wireless network can be prevented from interacting with the target device over Wi-Fi, and it also prevents ADB over Wi-Fi which is present on certain Android devices.

4 Underlying Cause For The Soft Reboot

The intents sent by the attacking app have the `FLAG_ACTIVITY_MULTIPLE_TASK` and `FLAG_ACTIVITY_NEW_TASK` flags set, so a new starting window with a new task stack will be required for each activity. In this section, the classes that end with "`Service`" are contained within the `system_server` process. The `com.android.server.wm.WindowManagerService` class [4] creates a window for the activity and each window requires a pair of `android.view.InputChannel` objects to be created so that the input events from the input device files can be delivered to the activity window. Third-party applications cannot read directly from the input device files which are contained in the `/dev/input` directory, but `system_server` has permission to read from them since it belongs to the `input` group. Therefore, `WindowManagerService` creates a pair of sockets using the `socketpair()` system call, registers the input channel with the window via the `com.android.server.input.InputManagerService` class, and transfers the output channel to the application. This allows the application to consume and process input events from the user via `system_server`.

A socket pair requires a file descriptor for each end of the socket pair. Each created activity will initially make `system_server` use two file descriptors. It will then transfer one socket to the attacking app, although during the attack `system_server` is processing a deluge of intents and does not get a chance to transfer the socket. This results in `system_server` using two file descriptors per activity created which makes `system_server` get closer to approaching the soft limit of 1,024 per-process file descriptors set by the kernel. Once the soft limit is reached, `system_server` cannot open or create any new files, pipes, or sockets, and `WindowManagerService` will fail to create the starting window for each activity.

The attacking app will encounter an uncaught exception once its activities cannot be created. The attacking app uses an `android.view.InputChannel` object received from the `WindowManagerService` as a parameter to the `android.view.InputEventReceiver` constructor. The `InputEventReceiver` object is used to queue the received user events so that they can be stored while waiting to be consumed by the application. The `InputChannel` object that the application received will be null. So an exception will be thrown by the `InputEventReceiver.nativeInit()` native method in the attacking application which goes uncaught and causes it to terminate.

When the attacking app crashes, `ActivityManagerService` tries to display an `android.app.Dialog` object indicating that the attacking app has crashed. A socket will be required to deliver the user input to the window of the `Dialog`

```
Intent i = new Intent(this, RebootMainActivity.class);
i.setFlags(Intent.FLAG_ACTIVITY_MULTIPLE_TASK | Intent.
    ↪ FLAG_ACTIVITY_NEW_TASK);

TaskStackBuilder tsb = TaskStackBuilder.create(this);
for (int a = 0; a < 1024; a++)
  tsb.addNextIntent(i);

if (Build.VERSION.SDK_INT >= 23) {
  while (true)
    tsb.startActivities();
} else {
  PendingIntent pi = PendingIntent.getActivity(getApplicationContext(),
      ↪ 0, i, PendingIntent.FLAG_CANCEL_CURRENT);
  AlarmManager am = (AlarmManager) this.getSystemService(Context.
      ↪ ALARM_SERVICE);
  am.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP, 1, 1, pi);
  tsb.startActivities();
}
```

Listing 1.2. Rapidly sending Intents using pending intents via AlarmManager and TaskStackBuilder, causing a soft reboot.

system message. `system_server` will not be able to create the socket, and an uncaught exception occurs. The `zygote` daemon process contains pre-loaded classes and resources and forks itself to create other applications quickly. `zygote` [6] starts `system_server` with the `--runtime-args` flag which provides the threads of `system_server` with an `UncaughtExceptionHandler` interface object of the type `com.android.internal.os.RuntimeInit.UncaughtHandler` [3]. It receives uncaught exceptions occurring within the threads of `system_server`. It only has one method and all of its code is within `try-catch-finally` blocks. The `finally` block calls the `android.os.Process.killProcess(int)` API call with an integer parameter that is the result of the `Process.myPid()` API call. Since the thread that has the uncaught exception occurs within `system_server`, this results in `system_server` both sending and receiving the `SIGKILL` signal, which results in its termination.

`zygote` is the parent process of `system_server`, so it will receive a `SIGCHLD` signal when `system_server` terminates. For each `SIGCHLD` signal that `zygote` receives, it will specifically check if the terminated child process is `system_server`. If `system_server` terminates, then `zygote` will send the `SIGKILL` signal to itself [5] which results in a soft reboot. The `init` process will then restart `zygote` since it is declared as a service in the `init.rc` file [2]. `zygote` will then restart `system_server`.

Listing 1.2 provides the source code to cause a soft reboot by rapidly sending intents. The attack uses `AlarmManager` to send an intent every millisecond in builds prior to Android 6. The `android.app.TaskStackBuilder` class is used to send 1,024 intents repeatedly for Android 6. The use of `TaskStackBuilder` requires

Android 4.1 or above. The `Service.startActivities(Intent[])` API call can be used in place of `TaskStackBuilder` which requires Android 3.0 or above.

5 Attack Evaluation

We tested the soft reboot cycle DoS attack on various Android devices. Some of the newer Android platforms tend not have safe mode and some do not have easy access to recovery mode, so we focused on these devices. All of these devices were running a non-rooted stock version of the Android OS that came pre-installed on the device. All of these devices had ADB over USB disabled by default. Table 1 aggregates the results of the experimental data.

Table 1. Test devices and results summary.

Device	Build No.	Android Version	Vulnerable	Recoverable
Sony Bravia XBR-43X830C TV	LMY48E.S63	5.1.1	Yes	No
Moto 360 1st Gen. Smartwatch	LDZ22O	5.1.1	Yes	Yes$^{\wr}$
Amazon Fire TV Stick 1st Gen.	JDQ39	4.2.2	Yes	No†
Xiaomi Mi Mini TV Box	KOT49H	4.4.2	No	Yes
Nvidia Shield Android TV	LMY47D	5.1	Yes	Yes‡
Amazon Fire 7″ Tablet	LMY47O	5.1.1	Yes	Yes‡
Devices prior to Android 4.1	-	<4.1	Yes	Yes‡

$^{\wr}$ Recovering requires crafting a special USB cable and flashing firmware images.
† Recovering requires ADB over USB, which is disabled by default, to be enabled prior to the attack.
‡ Recovering requires a *full* factory reset in recovery mode or flashing firmware images.

5.1 Sony Bravia XBR-43X830C Android TV

The Sony Bravia XBR-43X830C Android TV is vulnerable to the soft reboot cycle DoS attack, and there is no known way to recover. During our testing, the device was running Android 5.1.1 with a build fingerprint of `Sony/SVP4KDTV15_UC/SVP-DTV15:5.1.1/LMY48E.S63/2.473:user/release-keys`. The only way to perform a factory reset of the device is through the Settings app [14]. During the attack, the GUI becomes unresponsive to the infrared remote which prevents the user from reaching the Settings app to perform a factory reset. The device does have ADB over Wi-Fi, but this can be subverted since the attacking application disables Wi-Fi. This device does not have the ADB over USB capability. The device also does not have safe mode, recovery mode, or fastboot mode. Therefore, the user is unable to uninstall the application, perform a factory reset, or flash firmware images. Booting to fastboot mode via ADB over Wi-Fi will show a black screen, but it will also soft brick the device as it will not boot properly after that. The device comes pre-installed with Google Play so the user can download apps, and they can also be installed via ADB over Wi-Fi.

5.2 Moto 360 1st Generation Smartwatch

The Moto 360 1st generation smartwatch is vulnerable to the soft reboot cycle DoS attack, although there is a way to recover via a modified USB cable that can be used to unlock the bootloader and flash firmware images to the device [12]. During our testing, the device was running Android 5.1.1 with a build fingerprint of `motorola/metallica/minnow:5.1.1/LDZ22O/2006643:user/release-keys`. The device allows the user to directly install or uninstall apps using ADB over Bluetooth. When a user installs or uninstalls an app on an Android smartphone or tablet, which is paired with an Android Wear device, the accompanying Android Wear app, if present, will also be installed or uninstalled from the Android Wear device. The Moto 360 does not have a direct way to uninstall a particular application through its GUI. The user has about 8 seconds to perform some action on the device before the GUI becomes unresponsive. The user can initiate a factory reset through the GUI, but it will not have enough time to complete and be successful before the device soft reboots. The Moto 360 lacks a standard USB interface, so only ADB over Bluetooth is available. The attack app will disable Bluetooth to prevent communication with paired devices.

5.3 Amazon Fire TV Stick 1st Generation

The Amazon Fire TV Stick 1st generation is vulnerable to the soft reboot cycle DoS attack and can leave the device in an unusable state if ADB over USB is not enabled prior to the attack. The device runs Amazon Fire OS 3.0, which is a modified version of Android 4.2.2. The device we tested had a build fingerprint: `BRCM/montoya:4.2.2/JDQ39/54.1.2.2_user_122066120:user/release-keys`. If ADB over USB is enabled prior the attack, the user can list the installed third-party applications and uninstall them as the device is booting. The malicious application programmatically disables Bluetooth and Wi-Fi. This renders any paired devices ineffective and precludes ADB over Wi-Fi. There are no hardware buttons to force the device to boot into recovery mode or bootloader mode from a powered-off or booting state. This will effectively preclude the user from removing the application if ADB over USB is not enabled prior to the attack.

5.4 Xiaomi Mi TV Box Mini

The Xiaomi Mi TV Box Mini is not vulnerable to the soft reboot cycle DoS attack. The device we tested was running Android 4.4.2 and had a build fingerprint of `Xiaomi/forrestgump/forrestgump:4.4.2/KOT49H/566:user/release-keys`. Applications can be installed through the browser or a network-connected device. Communication with the device is performed via a Bluetooth remote, and it contains no USB interfaces. The device does not send the `BOOT_COMPLETED` broadcast intent to third-party applications, so the application is unable to soft reboot the device after the devices completes the boot process.

5.5 Amazon Fire 7″ Tablet

The Amazon Fire 7″ Tablet is vulnerable to the soft reboot cycle DoS attack if ADB over USB is not enabled prior to the attack. If ADB over USB is not enabled prior to the attack, then the user must perform a factory reset of the device or flash firmware images to the device. The device we tested was running Amazon Fire OS 5.0, which is a modified version of Android 5.1.1 and had a build fingerprint of `Amazon/full_ford/ford:5.1.1/LMY47O/37.5.4.1_user_541112720:user/release-keys`. The attacking app receives the `android.hardware.usb.action.USB_STATE` broadcast intent because it is sent prior to the `BOOT_COMPLETED` broadcast intent and does not require any permissions to be able to receive it. This broadcast intent is received by the attacking app prior to the Amazon launcher being displayed, so the user is precluded from uninstalling the app via the GUI. The device provides easy access to recovery mode from a powered-off state by holding the volume down and power buttons during boot.

5.6 Nvidia Shield Android TV

The Nvidia Shield Android TV device is vulnerable to the soft reboot cycle DoS attack if ADB over USB is not enabled prior to the attack. The device we tested was running Android 5.1.1 and had a build fingerprint of `NVIDIA/foster_e/foster:5.1/LMY47D/35739_609.6420:user/release-keys`. The device does not have safe mode and ADB over Wi-Fi can be programmatically disabled. The only way to recover is by performing a factory reset or flashing firmware images to the device. There is a method to perform a factory reset that is not published on Nvidia's website [1]. Alternatively, the user can access the fastboot menu and flash firmware images.

5.7 General Android mini PC Devices

Android mini PC devices are somewhat vulnerable to the soft reboot cycle DoS attack since they generally lack safe mode. Some devices allow the user to push a button during boot to enter recovery mode. In addition, some devices can utilize the SD card to flash firmware images to the device. Whether the attack is effective or not depends on the specific device and the mechanisms for recovery it provides.

5.8 Android Devices Prior to Android 4.1

Safe mode was introduced in Android 4.1. Prior to Android 4.1, the user was forced to perform a factory reset via recovery mode or flash firmware images to remove an application that persistently soft rebooted the device. According to the Android Dashboard, devices running a version of Android prior to Android 4.1 made up 5.0% of all Android devices as of March 7, 2016 [8].

6 Standalone Defense App

We developed an anti-reboot app (source available at [9]) that passively monitors intents sent by third-party apps on the system, and disables or uninstalls apps that attempt to flood the system with intents. The anti-reboot app observes intents by reading the system log buffer using `logcat` on the device, and parsing the log messages searching for intents. The app filters log messages using relevant log tags to reduce the amount of log messages it processes. For every observed intent, the sender's package name is logged and its total outbound intents count n is incremented. The anti-reboot app only considers intents that create new tasks, i.e., the `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_MULTIPLE_TASK` intent flags are set. It also ignores intents sent by system apps by filtering on the process User ID (UID) since system apps are assigned UIDs that are less than 10,000. Anti-reboot uses a one-level decay, where the intent count n is decreased by a constant c every second. This is intended to simulate the time a user would interact with a new activity before dismissing it. In other words, the value of c controls the tolerable persistence level of an offending app. For a period of t seconds, this results in an effective intent count $n' = n - ct$, and an effective sending rate $\rho = \frac{n'}{t} = \frac{n}{t} - c$. Finally, a monitored app is disabled or uninstalled if its corresponding n' exceeds a preset threshold (θ), which indicates that the monitored app has more than θ *active* task stacks.

6.1 Parameters Selection

There are two parameters that control the detection performance of the anti-reboot app: the intent decay c, and the cutoff threshold θ at which an app is disabled or uninstalled. The value of c controls the tolerance level of the defense to apps that persistently send multiple intents over time. While benign apps may create new tasks, such behavior typically lasts for only a very short period of time (i.e., short bursts) compared to attacking apps which need to be highly persistent in order to adversely affect the system. Therefore, the higher the value of c, the higher the tolerance and the more likely an attack may go undetected. A reasonable value of c would mimic the time it takes a user to click the recent tasks button and dismiss an activity off the screen, which takes about 2 seconds. Therefore, we set c to one intent every 2 seconds, i.e., $c = 0.5$.

Avoiding False Positives The cutoff threshold θ controls when an attack is detected, based on the number of active task stacks s the attack app has created. Note that $s \leq n'$, since each task stack would hold at least one activity. Since an attack is detected if $n' \geq \theta$, setting θ to a very small value may result in faster detection at the expense of false positives (i.e., false alarms). Conversely, a very large value of θ results in lower detection rate. We can pick a reasonable value of θ by estimating an upper bound on n' for *benign* apps. Recent studies (e.g., [17,20]) have shown that the total number of activities declared in an app's manifest is less than 110 for the top 30 apps in the market, with a total of 60 foreground activities created on the device *per day* from the top 800 apps on the

market. Therefore, we set $\theta = 200$, which allows 200 task stacks to be created at any point in time. This is more than three times the number (60) of task stacks that would be created, in the worst case, by benign apps if we assume each of the benign 60 activities was created in a new task stack and was never terminated.

In versions of Android earlier than 6.0, where `AlarmManager` does not have a minimum recurrence interval of 60 seconds, attacking apps can flood the system with activities using pending intents with short repeat intervals. To mitigate this, and in addition to observing intents, the anti-reboot app monitors the count and repeat interval of active pending intents being processed by the `AlarmManager`. It periodically retrieves a snapshot of the `AlarmManager` state by executing the `dumpsys alarm` command. Note that excessively running `dumpsys` can harm the overall system performance, while very long query periods can cause the attacks to go undetected. We empirically found that executing `dumpsys` every 500ms is suitable on the test devices used in this study. For each pending intent record, the anti-reboot app extracts the package name of the source app and the repeat interval. If the interval is less than a predefined threshold (set to 60 seconds as in Android 6.0), or the number of active pending intents of a source app is more than θ, the source app is flagged and is either disabled or uninstalled.

6.2 Detection Results

The anti-reboot app detected the soft reboot attack and identified the source of the attack 100% of the time during out testing, even when the attack was in its most aggressive form. In many cases, we observed that the device reboots before the anti-reboot app gets a chance to disable or uninstall the attacking app. This is mainly due to the fact that the attacking app can request to start up to $5,500$ new tasks in a single transaction using `Service.startActivities(Intent[])` API call. This quickly depletes the file descriptors of `system_server` which inhibits its capabilities and renders `system_server` unresponsive to any requests to disable or uninstall the offending app. To mitigate this, the anti-reboot app records the package name of offending apps along with a time stamp of when the attack was detected in persistent memory. It then checks when the system was soft rebooted, and if an offending app was detected within a 60 second period before the soft reboot, it disables the offending app after the soft reboot and informs the user. In addition, we confirm a soft reboot by checking to see if the Process ID of `system_server` has changed, which occurs during a soft reboot. The user can re-enable disabled apps through the GUI of the anti-reboot app.

We emphasize that it is not possible to rate-limit the intents sent by processes, without changes to the OS itself. Even then, a balance has to be struck between usability and security. If the system sets overly strict limits on the sending rate of intents, apps may become unresponsive or sluggish, resulting in an overall degradation of the system performance and user experience. In addition, it is not straightforward to implement rate-limiting in a system that is heavily event-driven such as Android. If the system decides to silently drop intents, apps are likely to malfunction as a result of lost intents. Notifying apps that they are exceeding the rate-limit would require a back channel from `system_server` to the

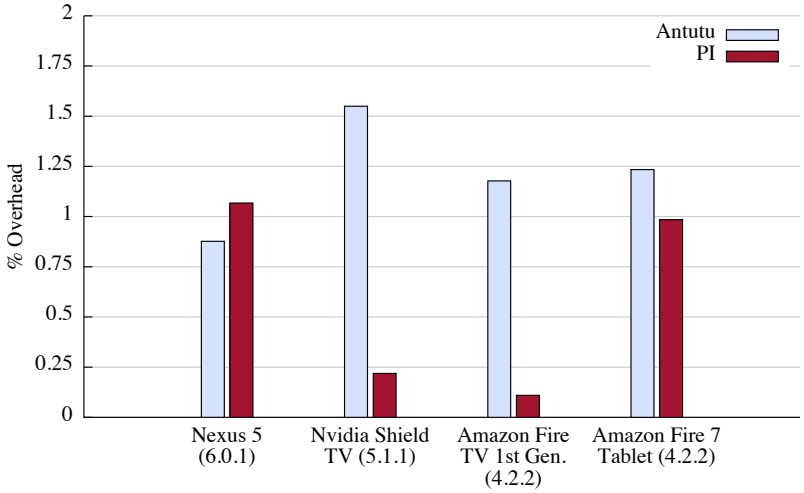

Fig. 2. Performance overhead based on AnTuTu Benchmark and BenchmarkPI scores.

app, besides requiring the app to anticipate and handle the notification, which further complicates the design of both the OS and the apps. We are unaware if this attack have been used in "the wild." After informing Amazon of the DoS attack, they created a detection mechanism for it in the Amazon AppStore. Google did not respond to our question whether or not the attack app would make it through their vetting process to be available on Google Play.

6.3 Performance Evaluation

We tested the overhead introduced by the anti-reboot defense app by using the following two benchmarks: AnTuTu Benchmark v6.0.1 and BenchmarkPI v1.1. AnTuTu Benchmark provides an aggregate score that combines both multitasking, user experience, CPU and memory speeds, and 3D rendering performance. BenchmarkPI is a CPU time benchmark that computes π to the n^{th} digit. We tested the defense app on the following devices: Nexus 5 running AOSP Android 6.0.1, Nvidia Shield Android TV running Android 5.1.1, Amazon Fire TV 1$^{\text{st}}$ generation running Android 4.2.2, and Amazon Fire 7$''$ tablet running Android 5.1.1. Under each scenario, we performed 20 runs and took the average of the resulting benchmark scores. We report the overhead as the percentage degradation in the aggregated average of the benchmark scores.

Figure 2 shows the overhead in the benchmark scores of AnTuTu Benchmark and BenchmarkPI. The overhead ranged from 0.8% to 1.51% for AnTuTu Benchmark and 0.14% to 1.15 for BenchmarkPI. The overhead from the defense app is mainly due to the threads it spawns to continuously monitor the Android log and process the output of the `dumpsys alarm` command to record intent usage and attribute them to the app that sent them. Overall, the defense app introduced

a small amount of overhead (less than 1.6%) which we believe is acceptable for the service it provides.

6.4 Framework Defenses

We suggest changes be made to the `ActivityManagerService` class in the Android framework to prevent a single app from starting an arbitrarily large amount of activities. Currently, the amount of intents that can be sent to be processed by `ActivityManagerService` is only limited by the Android Binder transaction buffer size. On Android 6, this enables an app to send a around 5,500 intents to be processed by `ActivityManagerService` in a single transaction using the `Service.startActivities(Intent[])` API call. A limit of less than 400 concurrent activities should be imposed on each app to preclude it from soft rebooting the device. Alternatively, a proper rate for rate-limiting of intents can be established from empirical analysis of intent usage among third-party applications. We recommend that once the user selects to perform a factory reset of an Android Wear device that all third-party applications should be terminated so they cannot attempt to interfere with the factory reset process. In addition, introducing some delay before sending the `BOOT_COMPLETED` broadcast intent and similar intents to third-party apps can provide the user additional time to perform a factory reset through the Settings application.

7 Related Work

Researchers have previously discovered methods to perform a soft reboot of Android devices. Armando et al. [16] discovered a vulnerability that made the device reboot by repeatedly forking processes from the `zygote` process from a third-party app. Huang et al. [25] discovered flaws in the concurrency control within `system_server`. When a monitor lock is held for more than a certain time threshold (i.e., 60 seconds), the watchdog process will terminate `system_server` since it appears that the process has encountered a deadlock. Terminating `system_server` results in a soft reboot of the Android OS. They developed a static tool to identify risky use of monitor locks within `system_server` so they can be triggered.

Chin et al. [21] presented various DoS attacks by intercepting intents destined for another application. This is due to apps using implicit intents by using an action that is declared in an application component's intent filter, as opposed to using the fully qualified class name of an application component. Intent hijacking can lead to the leaking of sensitive data sent embedded in an intent object. Johnson et al. [26] developed various DoS attacks on device resources and system availability using intent-based attacks. They discovered that a third-party application can monopolize the camera and microphone resources from a service application component running the background. The intent-based attacks can render the system unresponsive to the user, target and terminate other running applications, and soft reboot the device. We have continued this research and

proposed additional defenses and gathered experimental data by using the soft reboot cycle DoS attack on a range of Android devices.

Antunes et al. [15] proposed a system for testing server programs for resource exhaustion vulnerabilities by spraying the server with fuzzed inputs that are generated from a user-supplied specification of the server protocol. In [24], Groza et al. extends and formalizes the idea by formally modeling DoS attacks using cost-based rules that are dependent on the steps of the server protocol. Chang et al. [19] proposed a system that scans the source code of programs for potential code sites that may result in uncontrolled CPU time and stack consumption, and are influenced by untrusted input. Elsabagh et al. [22] proposed a system that models both the temporal and spatial information in resource consumption behavior of programs, and enforces the model at runtime. Extending such ideas to Android remains an open challenge, especially because of Android's uncoupled execution nature which heavily depends on inter-application communication.

8 Conclusion

By introducing a novel strategy for the soft reboot cycle DoS attack, we show that installing a third-party application, even with a limited set of permissions, can render certain Android devices unusable. In other cases, the user needed to perform a factory reset or flash firmware images to recover the victim device. Furthermore, we provide a detailed explanation as to the the underlying cause of the soft reboot that occurs in the Android framework. To support our claims, we reference the actual Android 6 source code and describe the mechanics of the attack strategy. To mitigate the attack, we leverage the existing Android framework to suggest changes that would either significantly reduce or eliminate the effects of the attacks. As a proof-of-concept, we implemented an open-source Android application that provides concrete countermeasures to prevent the attack and can be utilized by device manufacturers without modifying the device or the Android framework. As a final note, to ensure that our research is not misused, we informed Google and all of the affected device manufacturers listed in this paper so that Android devices can be made more secure.

References

1. Accessing SATV stock Recovery — nVidia Shield Android TV. `http://forum.xda-developers.com/shield-tv/general/accessing-satv-stock-recovery-t3300211`
2. Android Core Initialization Script. `https://android.googlesource.com/platform/system/core/+/android-6.0.0_r1/rootdir/init.rc`
3. Android Core Runtime Init. `https://android.googlesource.com/platform/frameworks/base/+/android-6.0.0_r1/core/java/com/android/internal/os/RuntimeInit.java`
4. Android Core Window Manager Service. `https://android.googlesource.com/platform/frameworks/base/+/android-6.0.0_r1/services/core/java/com/android/server/wm/WindowManagerService.java`

5. Android Core Zygote. https://android.googlesource.com/platform/frameworks/base/+/android-6.0.0_r1/core/jni/com_android_internal_os_Zygote.cpp
6. Android Core Zygote Init. https://android.googlesource.com/platform/frameworks/base/+/android-6.0.0_r1/core/java/com/android/internal/os/ZygoteInit.java
7. Android Debug Bridge | Android Developers. http://developer.android.com/tools/help/adb.html
8. Dashboards | Android Developers. http://developer.android.com/about/dashboards/index.html
9. endlessrecursion/antireboot: A standalone App to defend against reboot cycle DoS Attacks on Android. https://github.com/endlessrecursion/antireboot
10. Malware Uses SE Tricks to Enable Automatic App Installation. http://www.tripwire.com/state-of-security/latest-security-news/android-malware-uses-social-engineering-to-enable-automatic-app-installation/
11. Mi TV box Mini. http://xiaomi-mi.com/tv-box/xiaomi-mi-box-mini-tv-console/
12. Moto 360 adapter usb cable | How to Root Android. http://www.rootjunky.com/moto-360-adapter-usb-cable/
13. OmniRAT Takes Over Android Devices Through Social Engineering Tricks. https://securityintelligence.com/news/omnirat-takes-over-android-devices-through-social-engineering-tricks/
14. SONY | eSupport - How to reset the Android TV to factory settings. https://us.en.kb.sony.com/app/answers/detail/a_id/60594
15. Antunes, J., Neves, N.F., Veríssimo, P.J.: Detection and prediction of resource-exhaustion vulnerabilities. In: Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on. pp. 87–96. IEEE (2008)
16. Armando, A., Merlo, A., Migliardi, M., Verderame, L.: Would you mind forking this process? a denial of service attack on android (and some countermeasures). In: Information Security and Privacy Research, pp. 13–24. Springer (2012)
17. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 641–660. OOPSLA '13, ACM (2013)
18. Bhattacharya, P., Yang, L., Guo, M., Qian, K., Yang, M.: Learning Mobile Security with Labware. Security Privacy, IEEE 12(1), 69–72 (Jan 2014)
19. Chang, R., Jiang, G., Ivančić, F., Sankaranarayanan, S., Shmatikov, V.: Inputs of coma: Static detection of denial-of-service vulnerabilities. In: Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE. pp. 186–199. IEEE (2009)
20. Chen, X., Ding, N., Jindal, A., Hu, Y.C., Gupta, M., Vannithamby, R.: Smartphone energy drain in the wild: Analysis and implications. In: Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. pp. 151–164. ACM (2015)
21. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing Inter-application Communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services. pp. 239–252. MobiSys '11, ACM (2011)
22. Elsabagh, M., Barbará, D., Fleck, D., Stavrou, A.: Radmin: Early detection of application-level resource exhaustion and starvation attacks. In: Research in Attacks, Intrusions, and Defenses, pp. 515–537. Springer (2015)
23. Fedler, R., Schütte, J., Kulicke, M.: On the effectiveness of malware protection on android. Fraunhofer AISEC, Berlin, Tech. Rep (2013)

24. Groza, B., Minea, M.: Formal modelling and automatic detection of resource exhaustion attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 326–333. ACM (2011)
25. Huang, H., Zhu, S., Chen, K., Liu, P.: From system services freezing to system server shutdown in android: All you need is a loop in an app. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1236–1247. ACM (2015)
26. Johnson, R., Elsabagh, M., Stavrou, A., Sritapan, V.: Targeted DoS on Android: How to Disable Android in 10 Seconds or Less. In: Proceedings of the 10th International Conference on Malicious and Unwanted Software. pp. 239–252 (2015)
27. Liu, X., Yang, X., Lu, Y.: To filter or to authorize: Network-layer dos defense against multimillion-node botnets. ACM SIGCOMM Computer Communication Review 38(4), 195–206 (2008)
28. Peng, T., Leckie, C., Ramamohanarao, K.: Survey of network-based defense mechanisms countering the dos and ddos problems. ACM Comput. Surv. 39(1) (Apr 2007)
29. Potharaju, R., Newell, A., Nita-Rotaru, C., Zhang, X.: Plagiarizing smartphone applications: attack strategies and defense techniques. In: Engineering Secure Software and Systems, pp. 106–120. Springer (2012)
30. Vidas, T., Christin, N.: Sweetening Android Lemon Markets: Measuring and Combating Malware in Application Marketplaces. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy. pp. 197–208. CODASPY '13, ACM (2013)
31. Xiao, B., Chen, W., He, Y.: An autonomous defense against syn flooding attacks: Detect and throttle attacks at the victim side independently. Journal of Parallel and Distributed Computing 68(4), 456–470 (2008)
32. Yang, G., Gerla, M., Sanadidi, M.: Defense against low-rate tcp-targeted denial-of-service attacks. In: Computers and Communications, 2004. Proceedings. ISCC 2004. Ninth International Symposium on. vol. 1, pp. 345–350. IEEE (2004)
33. Yang, X., Wetherall, D., Anderson, T.: A dos-limiting network architecture. In: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. pp. 241–252. SIGCOMM '05, ACM (2005)
34. Zheng, M., Sun, M., Lui, J.: Droid Analytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on. pp. 163–171 (July 2013)
35. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy. pp. 317–326. CODASPY '12 (2012)
36. Zhou, Y., Jiang, X.: Dissecting Android Malware: Characterization and Evolution. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 95–109 (May 2012)