DEFINIT: An Analysis of Exposed Android Init Routines

Yuede Ji, *University of North Texas;* Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou, *Kryptowire*

https://www.usenix.org/conference/usenixsecurity21/presentation/ji

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

DEFINIT: An Analysis of Exposed Android Init Routines

Yuede Ji*
University of North Texas
yuede.ji@unt.edu

Mohamed Elsabagh
Kryptowire
melsabagh@kryptowire.com

Ryan Johnson
Kryptowire
rjohnson@kryptowire.com

Angelos Stavrou
Kryptowire
astavrou@kryptowire.com

Abstract

During the booting process of an Android device, a special daemon called Init is launched by the kernel as the first user-space process. Android allows vendors to extend the behavior of Init by introducing custom routines in .rc files. These Init routines can also be triggered by privileged pre-installed apps in a certain manner to accomplish privileged functionalities. However, as these pre-installed apps may fail to properly protect access to code sites triggering these Init routines, the capabilities of these routines may leak to unprivileged apps, resulting in crossing security boundaries set by the system. To this end, this study aims at investigating the prevalence of these Init routines and their security impact. We present DEFINIT as a tool to help automate the process of identifying Init routines exposed by pre-installed apps and estimating their potential security impact. Our findings are alarming. We found that custom Init routines added by vendors were substantial and had significant security impact. On a data set of 259 firmware from the top 21 vendors worldwide, we identified 1,947 exposed custom Init routines in 101 firmware from 13 vendors. Of these routines, 515 performed at least one sensitive action. We verified 89 instances spanning 30 firmware from 6 vendors, allowing unprivileged apps to perform sensitive functionalities without user interaction, including disabling SELinux enforcement, sniffing network traffic, reading system logs, among others.

1 Introduction

Android is open source and freely available to vendors to customize and port to different platforms. Owing to its open-source nature, Android has dominated the global smartphone market, holding more than 72% of the market share as of December 2020 [1]. The Android ecosystem is vast and versatile in supporting various platforms such as TVs, wearables, infotainment systems, and IoT devices. Android is built on top of a modified Linux with several changes at the kernel and user levels. Perhaps the most substantial of those is Android's process isolation and permission model: Android apps run in isolated processes, receive private storage spaces on the filesystem, can communicate using Android-specific secure inter-process communication (IPC) mechanisms, and require permission to access OS resources. This has also been the most studied aspect of Android from a security perspective [2–12].

Less studied are Android changes to Linux that are not necessarily visible to app developers and users. Few prior works have looked at the security risks stemming from Android customizations to boot loaders [13], kernel drivers [14], memory management [15], and SELinux policies [13, 16]. Other areas, such as changes to user-space daemons, received little to no attention. Of particular interest to us are changes made to the "Init" process, the first user-space process launched by the kernel after booting. Similar to Linux, Init on Android initializes the user space by mounting filesystems, initializing hardware, setting security policies, and loading essential system components. Different from Linux though, Init on Android is also the system property store where it keeps global system properties (in the form of key-value pairs) set by Init itself and other privileged Android processes.

More importantly, Android Init can execute custom routines in response to changing system properties. Android vendors can introduce privileged apps and executables to support certain vendor-specific hardware (e.g., sensors and custom partitions) and introduce value-added software services (e.g., custom pin-locked storage for vendor apps). These custom Init routines are defined in .rc files in the form of what Android Init calls "actions" and "services" using the Android Init Language [17] and execute with higher privileges than available to regular processes.[1]

Our work focuses on these vendor modifications to Init and attempts to assess their prevalence and potential security impact. Specifically, we are interested in studying security threats stemming from privileged apps exposing access to

*This work was done while the first author was interning at Kryptowire.

[1] Unless otherwise stated, in the rest of this document we use the term "Init routine" to collectively refer to Init actions and services.

Init routines that perform sensitive functionality. To this end, we propose an analysis system called DEFINIT to help us systematically analyze Android firmware images, map out the behaviors of custom Init routines, identify their necessary trigger conditions, analyze the privileged apps triggering them, and highlight sensitive routines exposed by privileged apps.[2]

We applied DEFINIT to 259 Android firmware from the top 21 vendors worldwide containing a total of 64,632 pre-installed apps and identified 1,947 exposed Init routines, all of which were added by vendors. Of these routines, 515 perform at least one sensitive action, impacting 101 firmware from 13 vendors. We further identified and verified 89 instances spanning 30 firmware from 6 vendors, allowing unprivileged apps to perform sensitive functionalities, such as disabling SELinux enforcement, capturing network traffic, reading system logs, recording the device screen, among others. Our findings highlight the significant security risks posed by vendor customizations to the Init process that are visible at the application layer, an area that has been previously unexplored. To summarize, we make the following contributions:

- **Novel System.** We propose DEFINIT, an automated practical system to process Android firmware images and identify Init routines, estimate their behavior, identify routines exposed by privileged apps, and highlight interesting routines that potentially pose a security risk.
- **Systematic Study.** We present the first comprehensive study on vendor customization to Android Init routines triggerable from privileged apps using a corpus of 259 firmware covering Android versions 8 to 11 from the top 21 vendors worldwide.
- **New Findings.** We provide new insights into the prevalence and security impact of customized Init routines and highlight multiple concrete exploitable instances with severe security and privacy impact to end users.

2 Background

2.1 Android Firmware Customization

We use the term Android firmware to refer to Android OS images that can be flashed to a device. An Android firmware contains all files necessary for the device to operate, and typically includes a bootloader, kernel, boot files, security policies, OS files, and pre-installed apps bundled by the device vendor. These files are packed as a set of partition blocks, and the firmware itself is delivered as a compressed archive (the exact file structure differs among vendors).

Android vendors customize their devices by including additional hardware and software to differentiate themselves by providing a unique, branded experience. The vendors take the official version of Android from the Android Open Source Project (AOSP) to make modifications and integrate their code. These modifications often touch many parts of the system, including boot files and OS components. It is possible to identify the base AOSP version a firmware image was forked from by inspecting the `/build.prop` file in a firmware root filesystem. Once identified, one can identify vendor customization by diffing files from a vendor firmware with their counterparts, if present, in the firmware base AOSP image.

Vendors also often include apps from their partners, hardware manufacturers, and carriers. Android apps are classified by type and an app's type limits the actions the app can perform on the device. A third-party app is an app that does not originate from the device vendor and is generally directly installed by the user through an app marketplace. A pre-installed app is an app that the vendor has included in the device firmware. Pre-installed apps are often necessary for proper system functionality. Pre-installed apps, by their nature of being selected by the vendor, can obtain permissions and capabilities that are not available to third-party apps. A pre-installed app in this regard is considered *privileged* versus third-party apps installed from the market.

2.2 Android Init

Like all Unix-like systems, Android has a special daemon process named Init (short for initialization) that executes first in user space once the kernel has finished booting. The Init process runs as `root` and acts as the progenitor to all other user space processes. The Init process is responsible for starting up the system, setting up directories and their permissions, mounting partitions, initializing peripherals, and setting up various system settings. On Android, the Init binary is located at `/init` at the root filesystem.

Android Init, however, diverges from traditional Unix-like systems in multiple ways. For instance, Init implements the system property store where it provides global read access to system properties to other processes (e.g., via the `getprop` command) and provides privileged processes with write access to system properties (e.g., via the `setprop` command). Android Init also acts as the device event handler (e.g., when the device is connected to USB).

Most importantly, device vendors can configure and extend the behavior of Android Init by defining custom Init routines in Init Resource Files (`.rc` files for short) that Init loads at boot time. These `.rc` files can be stored on different partitions, such as `/system` and `/vendor`. Init starts by loading the `/init.rc` file which further imports other `.rc` files. Init routines are implemented in the form of "actions" and "services" written in the Android Init Language [17]. An Android Init action is a named sequence of internal Init commands. An Init service specifies an external program for Init to launch, and potentially restart, with different runtime settings and security contexts.

Android Init can execute an Init routine at any point while the system is running when its corresponding "trigger" is

[2] DEFINIT stands for **D**etecting **E**xposed **F**unctionalities from **Init**.

matched. A trigger is a conditional statement that starts with "on" followed by a strictly conjunctive expression over Init event names (called Event Triggers) or system property values (called Property Triggers). Property Triggers use the word "property:" followed by a property name and expected value (e.g., "on property:service.adb.root=1"). Once the conditions for a trigger are satisfied at runtime, the associated actions for the trigger are executed.

Init recognizes several special property prefixes, including "ro.*" for read-only properties, "persist.*" for properties that survive reboots, and "sys.usb.*" for device USB attachment settings, among others. Init also recognizes two custom properties, "ctl.start=<service>" and "ctl.stop=<service>", that can be set by privileged apps to directly start and stop Init services without necessarily needing to satisfy their triggers.

Init property triggers use global system properties that can only be set by privileged apps and processes using internal Android commands and APIs, such as android.os.SystemProperties.set and setprop, that are not available to third-party apps. In this regard, privileged apps can be thought of as *deputies* as they can act on the request of an unprivileged app and invoke an Init routine by setting system properties. This can allow an unprivileged app to indirectly launch a sensitive Init routine through an open interface in a privileged app, resulting in crossing the security boundaries set by the system as the capabilities of the *exposed* Init routine are effectively leaked to the unprivileged app.

3 Threat Model and Assumptions

Our threat model assumes the users have installed an attacker-controlled, third-party app on their devices. This attack app will attempt to escalate its privileges by interacting with privileged pre-installed apps that have the capability to modify system properties that start Init routines.

We assume the attack app will interact with a privileged app by sending crafted inter-process communication (IPC) messages to exported (i.e., callable by other apps) components in the privileged app. This is a standard threat model specific to the Android ecosystem where apps are sandboxed and pre-installed apps can be granted permissions and capabilities not available to third-party apps [2–8, 10–12]. These methods allow the attack app to indirectly invoke code sites within a privileged app that cause the setting of system properties, launching an Init routine that performs a functionality that a third-party cannot perform given its limited privileges.

We only consider pre-installed apps as the access vector to Init routines. Analyzing other potential access vectors introduced by vendors is outside the scope of this work. Finally, we consider only Android versions 8.0 and higher since versions prior to 8.0 no longer receive system updates nor security patches as of this writing.

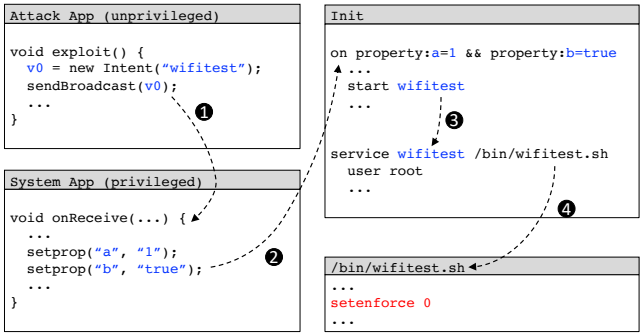

Figure 1: A simplified example based on a real-world case identified by DEFINIT for disabling SELinux enforcement through an Init service exposed by a pre-installed app.

4 Overview

4.1 Motivating Example

A real-world example of an exposed sensitive Init routine detected by DEFINIT is shown in Figure 1, where an unprivileged app disables SELinux policy enforcement on the device for all processes by exploiting a privileged app invoking a sensitive Init service. The figure shows the interactions between the attack third-party app, the privileged system app, the Init process, and the shell script invoked by a custom Init service to disable SELinux. A third-party attack app broadcasts a message (called an Intent in the context of Android) with an action of "wifitest" in step ❶. The Intent is received by an exported component in a privileged system app that registered to receive that action. Once received, the privileged app sets the system properties "a" to "1" and "b" to "true" in step ❷. These two system properties trigger an Init action (i.e., satisfy its conditions) that starts the wifitest service in step ❸. The wifitest service in turn executes a shell script /bin/wifitest.sh, in step ❹, as the root user. Finally, the script executes the setenforce 0 command that disables the system-wide enforcement of Mandatory Access Control (MAC) SELinux policies (the main defense mechanism Android systems depend on to establish mandatory privilege boundaries among processes).

4.2 Challenges and Key Insights

This study aims to identify potential security weaknesses stemming from Init routines exposed to unprivileged apps. We propose DEFINIT as a system that helps automatically highlight these potential issues for an analyst. DEFINIT has to handle multiple challenges concerned with processing Init files, understanding the behaviors of Init routines and their potential security impact, capturing dependencies and trigger conditions, identifying privileged apps invoking these routines, and detecting sensitive routines exposed to unprivileged actors. We discuss these challenges in the following.

C1: Processing Init files. While the Android Init Language is documented at [17], Init itself loads and processes `.rc` files dynamically in the presence of extra sources of information, such as system properties preloaded at boot time. Init `.rc` files can also reference Init sections defined in other files (using an `import` statement) and service definitions are polymorphic (i.e., a service can override its parent definition by using an `override` modifier). Since DEFINIT is static, we needed to implement a parser for `.rc` files that closely mimics the dynamism of Init. By studying the source code of Init, we found that we can start parsing at the root `/init.rc` file and nest into included files in depth-first order to mimic the behavior of Init. We discuss this in more detail in §5.1.

C2: Determining action and service behaviors. Init routines can execute programs represented as Init commands, ELF binaries, and shell scripts. DEFINIT needs to be able to determine the behavior of these programs to identify which routines perform security-relevant functionalities and the sensitivity of these functionalities. While the behaviors of individual commands and standard Android APIs are documented, the problem of automatically determining the behavior of arbitrary programs is undecidable as it can be reduced to the halting problem [18]. Nevertheless, the behavior of a program can be estimated based on information present in the executable program file that could indicate its behavior.

In DEFINIT, we estimate the behavior of routines by extracting code traces (including hardcoded parameter values) from their binaries and estimating the behavior of these traces using a compiled list of behaviors of potentially sensitive commands and standard Android APIs. This process maps a routine to a vector of behavioral categories, allowing an analyst to get an idea of its estimated general behavior. We also used static rules in our evaluation to help highlight specific behaviors by looking in the traces for certain call patterns. For example, if an Init service calls a system command to dump system logs to a file followed by a command that moves files to external storage, then it can be estimated that the service leaks the system logs to external storage. The specifics of behavior estimation vary depending on the kind of program executable being analyzed, which we detail out in §5.2.

C3: Modeling trigger conditions. There exist multiple interdependencies between Init actions, services, and Android commands and APIs. For instance, an action could start a service that runs a program that itself starts another Init service by setting an Init property to which a property trigger is registered. Actions could also start multiple services and commands. Trigger conditions can be composed of properties set by disjoint routines, making it difficult to identify the necessary trigger sequences to get Init to launch a certain routine. Therefore, DEFINIT needs to capture these dependencies (including transitive ones) to be able to reason about the conditions necessary to trigger a certain behavior via Init routines. To capture these interdependencies, DEFINIT builds a graph that we refer to as Init Dependency Graph (IDG). In an IDG, nodes represent Init triggers, Init services, and executables called by routines. (An executable here can be an Init command, a shell command, or an ELF binary.) Edges in an IDG represent call edges between triggers, services, and executables; and conjunction relationships between conjuncts in a composite property trigger. Using an IDG, we can efficiently extract relationships between Init actions, services, triggers, and the conditions necessary to execute a certain Init action or service. We discuss IDGs in §5.3.

C4: Identifying exposed routines and behaviors. On Android, Init properties can only be set by privileged apps (including privileged native binaries) by using internal APIs, such as `android.os.SystemProperties.set` and `___system__property__set`, that are not available to third-party apps. Privileged apps here can be thought of as deputies that control access to Init routines. Privileged apps that invoke sensitive Init routines based on requests from unprivileged apps can be subject to confused-deputy attacks where the capabilities performed by Init leak to the unprivileged apps.

Identifying privileged apps invoking Init routines requires identifying app call sites that invoke APIs setting system properties, and resolving the parameter values of these APIs to identify the properties being set and their corresponding values at each call site. In DEFINIT, we developed a technique to identify the property keys being read/written and the mapping between each property key and its corresponding value in a context- (i.e., taking the callee stacks at each relevant API call site into consideration) and flow-sensitive (i.e., taking statements execution order into consideration) manner. We then identify exposed routines by looking for control-flow paths from exported app entry points to relevant code sites. We tune our analysis to avoid false positives from dynamic bytecode constructs (e.g., virtual calls) at the expense of soundness (i.e., missing some valid flows). More details are provided in §5.4.

5 Detailed Design

Figure 2 shows the workflow of DEFINIT. Given an Android firmware image as input, DEFINIT unpacks it to extract needed files. It then processes Init `.rc` files to identify custom Init routines. We consider a service to be custom, i.e., a result of vendor modification, if it references an executable that was not present in the base AOSP image of the firmware. We consider an action as custom if its trigger is not found in the base AOSP image. DEFINIT then estimates the behavior of these routines and assesses their security impact. Following that, DEFINIT identifies privileged apps exposing access to these sensitive routines and generates a report containing a listing of the exposed routines, their estimated behaviors and

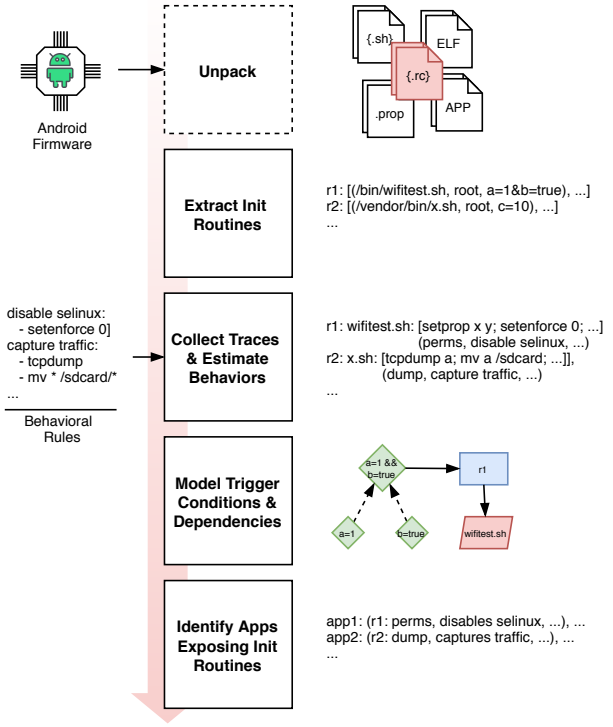

Figure 2: Workflow of DEFINIT.

security impact, privileged apps exposing them and how, and Init trigger conditions needed to invoke these routines. We discuss the details of these steps in the following.[3]

5.1 Extracting Init Routine Definitions

DEFINIT processes `.rc` files to extract Init routines and the commands or executables they invoke. Parsing occurs according to the syntax of the Android Init Language [17] in a way that mimics the runtime behavior of Init. Specifically, DEFINIT parses each `.rc` file line by line, starting at the root `/init.rc` file, then nests into imported files in depth-first order following the same import rules in [17, sec. imports]. Variables encountered during parsing of static Init constructs (e.g., `import` paths) are substituted with their corresponding default values from `.prop` and boot environment files.

Since an Init service definition can override a previous definition associated with the same service name, DEFINIT only keeps the most-specialized service definition that uses the `override` modifier (i.e., the last encountered definition in Init `.rc` parsing order that sets the `override` modifier). For a trigger that is declared multiple times, DEFINIT appends all its actions under the first-encountered trigger (this is equivalent to Init sequentially invoking the actions of each declaration of the trigger at runtime). The output of this step is an enumeration of the effective set of Init routines (i.e., the subset of routines that are *live* at runtime) and their associated triggers, SELinux modifiers, and other attributes as defined in [17].

[3] We omit the details of the firmware unpacking process as we employ standard unpacking tools and techniques. Interested readers can refer to prior work (e.g., [9, 11]) for information on unpacking techniques.

5.2 Estimating Behaviors of Init Routines

The goal of this step is to estimate the behavior of executables invoked by Init when a trigger is fired. Init can invoke three kinds of executables in response to a trigger: Init actions, shell scripts, and ELF binaries. We discuss how we collect code traces from each executable kind in the following.

Init actions are defined by the Init Language [17] as a named sequence of predefined Init commands, therefore DEFINIT extracts Init action traces from the action definitions in `.rc` files as a list of Init commands, substituting hardcoded property values as needed from `.prop` files.

For shell scripts, DEFINIT employs a custom shell tracer that dry-runs shell scripts inside a sandbox built on top of Bash trace mode (see `bash -x` option at [19]) to collect their command traces. Since these scripts are executed in a foreign environment, it is expected that they would incur runtime errors due to missing dependencies from their execution environment. Therefore, DEFINIT needs to carefully control their execution to maximize coverage. Specifically, DEFINIT taints environment and command-line arguments available at a shell script invocation site in an `.rc` file, and evaluates only conditional statements in the script that depend on (i.e., directly uses or derived from) these arguments. Additionally, DEFINIT ignores "sleep" statements and masks return codes of invoked shell commands to avoid prematurely exiting the script due to missing commands.

For ELF binaries, DEFINIT collects static traces of called APIs by traversing simple paths in the binary inter-procedural control-flow graph (ICFG) in depth-first order, starting at the binary entry point function and ignoring control flows through basic blocks not calling any APIs. For relevant APIs with potentially sensitive arguments, DEFINIT performs inter-procedural Def-Use analysis to propagate constant character strings and numerical definitions to API call sites to identify arguments at each call site of interest. In addition, DEFINIT extracts strings from the binary that resemble system commands by matching the first token of strings to executable file names and paths available in the input ROM. This whole process is done recursively through the ELF executable and its dynamically linked functions.

DEFINIT then uses the traces for each Init routine to annotate the routine with behavioral categories based on the curated list of behaviors of security-sensitive APIs and commands shown in Table 1. We collected these by, first, automatically enumerating all the commands in AOSP images and the APIs in Bionic libc. This resulted in 473 commands and 4,259 APIs. Then, we filtered out the obviously non-security-relevant ones, leaving us with 137 commands and 64 APIs.

Table 1: Security-sensitive APIs and commands used by DEFINIT for highlighting security-sensitive Init routines.

Category	APIs/Commands	Count
Device Settings	hid ime locksettings settings svc	5
Sensitive Data	atrace bugreport content diag_klog diag_mdlog diag_socket_log diag_uart_log dumpstate dumpsys logcat ramdump record_stream_new screencap screenrecord tcpdump	15
Networking	dnsmasq ifconfig iptables telecom send sendfile sendfile64 socket_local_server_bind	8
Package Management	applypatch pm dpm insmod patchoat	5
Permission Control	keystore appops setsid load_policy setenforce	5
Power Management	thermal_engine ___reboot android_reboot reboot	4
Process Management	cmd killall killpg ptrace service	5
UI Interaction	virtual_touchpad am input sendevent monkey uiautomator	6
Total		53

Finally, we consulted the documentations of these commands and APIs and shortlisted the potentially sensitive ones.

For each Init routine, DEFINIT annotates it with the counts of security-sensitive commands and APIs it executes. This categorization gives an analyst a basic understanding of the overall behavior of a service and its potential security impact. DEFINIT then uses pattern-matching rules to identify call sequences in the traces that indicate more specific interesting behaviors. For example, a common source of vulnerabilities in Android is leaking sensitive data to external (shared) storage, which DEFINIT can identify by looking in the traces for calls to a command from the Sensitive Data category followed by calls to commands that move files to a path on external storage. We developed 116 rules to match specific call sequences and parameters (one to three calls per rule). These rules are incrementally developed by analysts as they require domain knowledge of security weaknesses that may manifest as a result of invoking Android commands and APIs.

5.3 Modeling Trigger Conditions

To capture trigger conditions of Init routines, we propose a directed heterogeneous graph structure that we refer to as Init Dependency Graph (IDG) in which we encode trigger conditions and transitive dependencies between Init triggers, actions, services, and executed programs. Figure 3 shows the IDG for the running example in Figure 1. DEFINIT uses an IDG to identify what functionality Init performs in response to properties set by privileged apps and binaries.

There are three types of nodes in the IDG: trigger, service, and executable nodes. A trigger node represents a single Init trigger condition. For example, the trigger node `"a=1"` denotes that the property key `"a"` needs to equal `"1"` at that state in the IDG. We split composite triggers (boolean conjunctions of trigger conditions) into multiple nodes, one for each trigger condition (a conjunct). A service node represents an Init service. An executable node is a terminal node that represents the executable invoked by an Init command (as part of an Init action) or a service. We use one unique node for each unique executable invocation (including the executable arguments). A trigger node is also added for the custom Init property `ctl.start=<service name>` for each identified service.

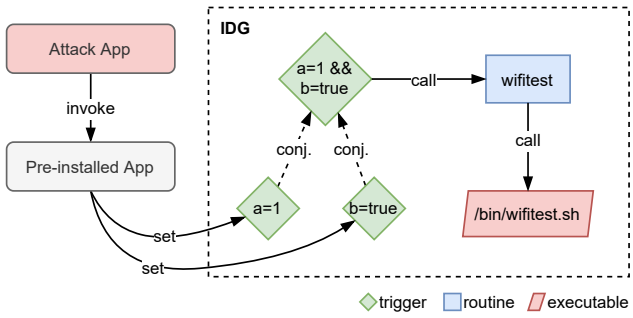

Figure 3: Simplified Init Dependency Graph (IDG) for the running example in Figure 1.

Our IDG construction algorithm is shown in Algorithm 1. Edges in an IDG can be call edges or conjunction edges. A call edge represents a caller-callee relationship between different nodes. Note that a trigger node can call other trigger nodes by setting properties or triggering events using Init commands such as `setprop` and `trigger`. Also, Init actions can start services and invoke executables using Init commands such as `start` and `exec`.

Conjunction edges in an IDG encode the dependency of a multi-condition trigger (a boolean conjunction) on its individual operand conditions (each is a property trigger node). For example, the trigger `"a=1 && b=true"` in Figure 1 will have two conjunction edges from the trigger nodes `"a=1"` and `"b=true"`. Note that a conjunction trigger can only be satisfied when all its operand property conditions are satisfied, and a property can be used by different trigger conditions (potentially with different property values).

Finally, we add fall-through call edges from executables that call an Android API or a command setting an Init property to the corresponding target trigger nodes that use that property. The property keys and values in these scenarios are extracted from the traces collected in §5.2.

DEFINIT builds one IDG for each firmware image. The IDG provides a global view of the transitions occurring inside Init that involve triggers, services, and executables in the firmware, allowing DEFINIT to understand what behaviors Init can launch and the conditions needed to trigger them by traversing the IDG as explained in the following section.

5.4 Identifying Exposed Routines

The next step for DEFINIT is to identify the mapping between privileged apps and sensitive Init routines. To do this,

Algorithm 1: Construct Init Dependency Graph.

inputs : $T \leftarrow$ Map of triggers to their routines
output : $G \leftarrow$ Init Dependency Graph

foreach *trigger* $t \in T$ **do**
  add node t to G if missing
  if t *is a conjunction trigger* **then**
    foreach *conjunct* $t_i \in t$ **do**
      add node t_i to G if missing
      add edge $t_i \xrightarrow{conj.} t$ to G
  foreach *action or service* s *in* $T[t]$ **do**
    add node s to G if missing
    add edge $t \xrightarrow{call} s$ to G
    foreach *executable* x *called by* s **do**
      add node x to G if missing
      add edge $s \xrightarrow{call} x$ to G
      foreach *property/service* p *set/called by* x **do**
        add node p to G if missing
        add edge $x \xrightarrow{call} p$ to G

Algorithm 2: Extract written properties.

inputs : $A \leftarrow$ {APIs to write a local/system property}
  $S \leftarrow$ app ICFG with Def-Use information
output : mapping between written keys and corresponding values

// S, K are in depth-first order
foreach *statement* $s \in S$ *calling some API* $\in A$ **do**
  $K \leftarrow$ {definition points in S of property keys used by the first argument at s}
  foreach $k \in K$ **do**
    foreach *call stack* T *carrying* k *to* s **do**
      $V \leftarrow$ {property values defined in the scope of T used by the second argument at s}
      emit s, k, V

DEFINIT first scans each pre-installed app for code sites that call certain Android APIs of the form `set(key, value)`, such as `android.os.SystemProperties.set`, to set a system property. Then, DEFINIT builds an ICFG and performs Def-Use based analysis to identify each set property key and its corresponding values in a context- and flow-sensitive manner where the keys and values are extracted per each call stack ending at a relevant API call site. Algorithm 2 shows the basic working principle of this technique. The goal here is to extract each key and its corresponding values set by an app along each call stack of a relevant API call site, rather than extracting bags of all keys and all values used at the call site.

Similarly, DEFINIT also extracts system properties read by privileged apps and local properties read/written by privileged apps that share the same UID. DEFINIT then adds corresponding nodes and edges to the IDG to capture indirect information flows between apps using these properties. This is necessary since Android allows apps to share process-scope properties by using the same process UIDs.[4] For example, a privileged app can have an exposed call site that sets a local property to signal another privileged app to invoke a sensitive Init routine. Not accounting for these cross-app properties would leave exposed routines hiding behind these indirections undetected. DEFINIT uses identified written properties to walk the IDG and discover sensitive Init routines that can be triggered. Specifically, for each privileged app, DEFINIT walks the IDG starting at property trigger nodes corresponding to properties written by the app (in depth-first order) and aggregates the behaviors of terminal executable nodes of traversed paths. When a conjunction node is reached, DEFINIT only traverses past it if all the conjuncts have been satisfied.

Finally, to mark privileged apps exposing Init routines, DEFINIT performs control-flow reachability analysis similar to [10, 11] by finding a control-flow path from any exported [20, ch. 4] entry point of an identified privileged app that invokes an Init service to the corresponding call site that results in invoking the service.

6 Evaluation and Security Impact

To understand the prevalence and impact of exposed Init routines, we performed a large-scale study using DEFINIT on 259 stock Android v8.0 to v11.0 firmware images covering 21 of the top vendors worldwide as shown in Table 2. These images contained a total of 64,632 pre-installed apps with an average of 262 apps per firmware. At the time of writing, Android v11.0 was recently released and only a few vendors provided public Android v11.0 images.

Table 2: Summary of tested Android firmware images.

Version	Vendors	Firmware (#/vendor)	Apps (#/firmware)
8	19	93 (2;10;5)	18,988 (57;805;211)
9	17	75 (1;6;4)	16,809 (148;452;229)
10	14	75 (1;11;5)	23,117 (18;504;269)
11	5	16 (1;4;3)	5,718 (193;527;339)
Total	21	259 (1;11;4)	64,632 (18;805;262)

Counts are 'total (min;max;avg)'

6.1 Prevalence of Custom Init Routines

Table 3 provides summary statistics of Init routines DEFINIT identified in the analyzed images. Of the tested 259 firmware, there was a total of 58,523 Init routines (223 per firmware). Among these, 38,846 (66%) were custom routines added by vendors. This averages to about 133 custom Init routines per firmware, with some vendors adding as many as 360 custom Init routines over AOSP. This shows the great extent to which

[4] Privileged apps can choose their UID by setting a special attribute in their manifest files, see https://developer.android.com/guide/topics/manifest/manifest-element#uid for details.

Table 3: Custom and exposed Init routines prevalence per Android version. All exposed routines were custom, added by vendors.

Version	Total	Custom	Exposed	Exposed Sensitive
8	15,602 (39;355;166)	8,613 (9;210;91)	305 (0;42;3)	81 (0;15;1)
9	16,719 (188;522;220)	8,537 (41;356;112)	600 (0;64;8)	166 (0;16;2)
10	21,558 (144;527;287)	12,576 (22;360;167)	911 (0;67;12)	244 (0;14;3)
11	4,644 (179;511;273)	2,704 (29;360;159)	131 (0;46;8)	24 (0;7;2)
Total	58,523 (39;527;236)	38,846 (9;360;133)	1,947 (0;67;8)	515 (0;16;2)

Counts are 'total (min;max;avg per firmware per version)'

vendors customize the Init process to introduce new functionalities. To put this in perspective, AOSP has about 130 Init routines on average, meaning that vendors introduce *at least* as many Init routines as already present in AOSP.

Figure 4 shows the distribution of nodes and edges in the IDGs constructed by DEFINIT from the firmware in our data set. On average, more than 50% of the firmware had at least 600 IDG nodes, 500 edges, and 300 different paths from a trigger to an executable with a path length of three or higher (i.e., three levels of indirection within Init from the moment a trigger is set until an executable is launched). This shows the complexity of the data- and control-flow facts encoded in the `.rc` files, which substantiates the need for an automated system like DEFINIT to bring interesting cases to the surface.

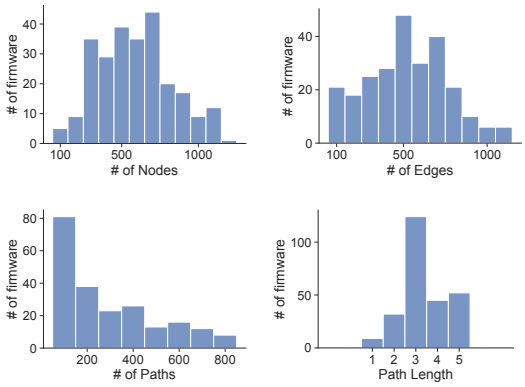

Figure 4: Distribution of IDG nodes, edges, and paths from a trigger node to a terminal executable node in the identified Init routines.

6.2 Characteristics of Exposed Routines

Of these Init routines, 1,947 were exposed by an IPC entry point of a privileged app. This averaged to about eight exposed Init routines per firmware. 515 of these exposed Init routines executed at least one sensitive command from those listed in Table 1. Interestingly (though unsurprisingly given the number of new routines added by vendors), all exposed routines were custom routines added by vendors. Notably,

Table 4: Top 10 process UIDs of identified Init routines. More than 90% of exposed routines were running as `system` or `root`.

UID	Total	Custom	Exposed	Exposed Sensitive
`root`	15,597	11,935	1,342	354
`system`	17,305	10,293	272	2
`default (root)`	9,764	3,146	188	113
`shell`	1,255	858	50	0
`bluetooth`	609	499	45	45
`logd`	599	99	45	0
`graphics`	351	96	3	0
`wifi`	1,563	603	1	1
`radio`	2,247	997	1	0
`nfc`	178	159	0	0
other	9,055	3,745	0	0
total	58,523	32,430	1,947	515

firmware images from vendors closest to AOSP (i.e., Google) had no exposed Init routines.[5]

Tables 4 and 5 list the process UIDs and SELinux domains of identified routines. More than 90% of the exposed routines ran as `root` or `system`, the two most privileged users on Android. Likewise, the majority ran under the default SELinux domains as well (`init` and `vendor_init`). All identified exposed routines had UIDs and domains that are significantly more privileged than those assigned to third-party apps by the system (randomly generated at install time and falls in the `untrusted_app` SELinux domain). It is unclear to us why exactly all these custom services needed to run with these privileged defaults. It appeared as if vendors simply followed the path of least resistance by using the defaults rather than properly using Init modifiers to confine their custom Init routines such that they have access only to the resources and capabilities necessary for their operation. Note that while there may be SELinux transition rules based on the file names of routine executables, these rules would not block exploitation of an exposed routine since it is unlikely that a vendor would add custom routines to invoke custom executables yet fail to configure SELinux transitions to allow the executable to function as intended. We discuss this further in §7.

The breakdown of shell scripts and binaries invoked by the identified Init routines is shown in Table 6. In total, 2,685

[5] We observed routines that were exposed only via exported GUI entry points, making them potential targets for GUI cloaking attacks [12]. We provide measurements on these in Appendix A.5.

Table 5: Top 10 SELinux domains of identified Init routines. The domain `init` is the default domain. The domain `vendor_init` is the default for routine executables located in the `/vendor` partition.

Domain	Total	Custom	Exposed	Exposed Sensitive
`init`	27,739	11,816	988	252
`rutilsdaemon`	2,600	2,545	632	204
`vendor_init`	24,497	15,719	102	15
`dumpstate`	183	176	50	12
`junklog`	90	90	29	2
`logserver`	133	66	28	2
`xlogcat`	25	25	18	0
`kapd`	96	96	14	0
`logoem`	32	32	12	12
`glogcat`	18	0	12	0
`other`	3,110	1,865	62	16
Total	58,523	32,430	1,947	515

Table 6: Executables used by Init routines in the analyzed firmware.

Type	Custom	Count
Total used		
Script	•	2,685 (1;100;10)
Script		310 (1;9;1)
Binary	•	16,772 (1;136;64)
Binary		18,419 (1;131;70)
Used by an app-triggered Init routine		
Script	•	1,606 (1;68;6)
Binary	•	863 (1;12;3)
Binary		504 (1;9;2)
Used by an exposed Init routine		
Script	•	1,161 (0;57;4)
Binary	•	414 (0;9;2)
Binary		181 (0;8;1)
Exposed and calls a sensitive command/API		
Script	•	581 (0;34;2)
Binary	•	410 (0;9;2)
Binary		82 (0;4;0.3)

Counts are 'total (min;max;avg per firmware)'

unique custom scripts (10 per firmware), 310 known scripts, 16,772 custom binaries (64 per firmware), and 18,419 known binaries were used by Init routines. Of these, 1,606 custom scripts, 863 custom binaries, and 504 known binaries were used by Init routines triggered from pre-installed apps. With regard to exposed routines, 1,161 custom scripts, 414 custom binaries, and 181 known binaries were called by at least one exposed Init routine. On average, there were four scripts and two binaries invoked by an exposed Init routine per firmware. Of those exposed scripts and binaries, 581 custom shell scripts and 410 custom binaries invoked at least one sensitive command. These numbers show the significant changes vendors introduce to the Init process. The results also suggest that vendors are more likely to use custom shell scripts rather than binaries for their custom app-triggered Init routines, probably due to the ease of developing shell scripts.

6.3 Impact of Exposed Behaviors

Table 7 shows the breakdown of exposed sensitive routines, grouped by behavior category. We discuss the overall prospect of these routines below and provide the full breakdown per sensitive command in Table A.6. Our discussion here is focused on exposed routines that can be abused without user interaction with the pre-installed apps exposing the sensitive functionalities. We report on routines that require user interaction in Appendix A.5.

Behaviors common to multiple vendors were for routines calling commands from the sensitive data, networking, and process management categories. The top category in terms of exposed routines was for intrusive routines accessing sensitive device data, such as memory dumps, system logs, and network traffic captures. This totaled to 336 unique routines across 65 firmware from 11 vendors. These routines were exposed by 109 different pre-installed apps. The majority (298) of these routines were triggered directly (i.e., via `ctl.start=<service>`) whereas 38 were triggered by setting system properties satisfying a trigger. The impact of these exposed routines accessing sensitive data can be significant if this data is transferred by the routines to shared storage on the device, making it accessible to all apps (privileged or not). These routines tend to expose user data through various debugging and development mechanisms such as capturing detailed log messages from all processes and dumping the state of all Android framework services.

For the 65 routines in the networking category, the majority were routines creating local domain socket servers. These local domain socket servers provide a communication interface for other processes on the device to interact with the server process, potentially introducing security weaknesses. Unfortunately, identifying the behavior lying behind the server socket process is a very challenging task that requires knowledge of the specifics of the protocol implemented by the involved processes. Nevertheless, various instance have been discovered in the wild where missing authorization checks in domain socket server processes has resulted in critical code and command injection vulnerabilities in privileged system processes [21–24].

Of the 30 routines in the process management category, the majority were delegating to Android framework services via the `service` command. The `service` command was generally used in conjunction with debugging routines to dump a snapshot of the active framework services on the system using the `service list` command. An interesting case is the `aee-reinit` Init service that uses the `ptrace` command. Upon manual inspection of the binary implementing the Init routine, it appeared to be a process that attaches to a target process using the `ptrace` command to dump its state, which can result in information leakage as third-party apps are not allowed to call `ptrace` on other processes on the system.

The device settings category had 13 identified routines to modify device-wide settings. The `svc` command was the most

Table 7: Init routines exposing sensitive functionalities, the number of apps exposing them, and the impacted firmware. Multiple matches in the same category are counted only once per unique routine.

Category	Total	Direct	Indirect	Apps	Firmware	Vendors
Device settings	13	13	0	13	13	2
Sensitive data	336	298	38	109	65	11
Networking	65	16	49	65	62	9
Package management	9	0	9	9	9	3
Permission control	9	8	1	9	9	2
Power management	6	0	6	5	5	3
Process management	30	25	5	28	28	5
UI interaction	48	29	19	48	42	6
Total (unique)	430	323	107	173	101	13

commonly used among these routines. Certain pre-installed apps used the `svc power reboot` command to reboot the device which can be repeatedly initiated by an adversary to prevent the user proper access to the device.

For routines in the package management category, the most sensitive behavior was loading a kernel module which was detected in nine firmware. We found a particularly interesting case where the kernel module was loaded from a writeable path, which may result in a third-party app being able to overwrite the kernel module and achieve arbitrary code execution in kernel space. Most of the other cases, based on their routine names, appeared to be for sniffing network packets.

The permission control category had only nine exposed Init routines, but this category contains some of the most sensitive commands. Specifically, the `setenforce 0` command disables SELinux, essentially exempting *all processes* from their Mandatory Access Control policies, allowing them to perform actions that would otherwise be blocked such as setting system properties, accessing sandboxes, and connecting to restricted framework services. Surprisingly, we found seven such instances in six firmware (one can be exploited without user interaction, six by clicking a button) all from one popular vendor from the largest manufacturer group globally, where SELinux can be disabled through an exposed Init routine.

The six routines in the power management category pertained to commands that initiate a reboot of the system. These may appear uninteresting, although they can be leveraged by an attacker to perform controlled DoS attacks by continuously rebooting the system, which, for example, can be leveraged in ransom DoS [25]. Of these six routines, five allowed an app to perform a programmatic reboot spanning three different vendors. This can also result in factory resetting the device and erasing all user data in certain cases [11].

The UI interaction category had 48 Init routines which were generally used to send IPC messages using the `am` command. All of the messages were implicit, lacking a named destination, except for few messages for opening the results of an operation in the default HTML viewer (i.e., `com.android.htmlviewer`). We observed some routines injecting key events for the "power" and "menu" buttons in the foreground device UI using the `input` command. While this likely has low impact, injecting these events may still cause undesirable effects on the system when used at inopportune times. Moreover, uncovering additional key events can be used to build a UI interaction toolkit for use by an attacker.

6.4 Vulnerability Studies

We further inspected routines that exhibited more specialized behaviors and manually inspected them to verify their potential security impact when triggered by an attacker. Table 8 shows the outcome of this analysis. Thus far, we have manually verified 89 vulnerabilities in 34 unique apps from 30 firmware from 6 vendors. Our disclosure process is still ongoing, and three vendors so far have confirmed our findings (covering 49 flaws in 11 firmware). Again, we only focus here on vulnerabilities that can be exploited without user interaction with the pre-installed apps. There are another 134 vulnerabilities in 52 unique apps from 35 firmware from 9 vendors that can be exploited but require user interaction with the pre-installed apps, which we outline in Table A.5.

Verification Methodology. The verification involved manually verifying the reported code paths to ensure the following: (1) There are no runtime checks (e.g., dependencies on UID, permissions, signatures, package names) on the path that may increase the attack requirements beyond what is accessible to a third-party app. (2) The privileged app sets the expected system properties to the required value. (3) The system properties trigger the expected Init routine. (4) The executable, corresponding to the triggered Init routine, performs the reported security-sensitive functionality.

For the stock Android devices we were able to obtain, we manually developed exploits to dynamically verify 53 findings (none requiring user interaction besides installing and running our attack app). Note that dynamically verifying all findings on their corresponding native Android devices presents significant difficulty since it requires purchasing Android devices for each vendor/model/version combination.

Table 8: Verified vulnerabilities and the functionalities they allow an unprivileged attack app to perform programmatically via inadequate access control exhibited by pre-installed apps.

Impact	Total	Apps	Firmware	Vendors	Versions
Read system logs	11	11	11	3	8,9,10
Record screen	5	5	5	1	8,9
Sniff modem traffic	7	7	7	2	8,9
Sniff Wi-Fi traffic	8	8	8	2	8,9
Sniff Bluetooth traffic	2	2	2	1	8,9
Read Wi-Fi passwords	3	3	3	1	9,10
Read dumpstate	7	7	7	2	8,9
Read dumpsys	10	10	10	3	8,9,10
Read kernel logs	10	10	10	3	8,9,10
Read bugreport	3	3	3	2	9
Read radio logs	6	6	6	2	9
Load kernel module	9	9	9	1	8,9
Disable SELinux	1	1	1	1	8
Reboot device	5	5	5	3	9,10
Write to node device	2	2	2	1	8
Total (unique)	89	34	30	6	8,9,10

Findings. In addition to disabling SELinux, loading kernel modules, and rebooting the device, we found instances where third-party apps can indirectly obtain the following data due to exposed Init routines: system logs (main log, kernel, radio), screen captures, telephony data (SMS messages, calls), extensive system dumps (dumpsys, dumpstate, bug reports), and packet captures (modem, Wi-Fi, Bluetooth). Overall, the impact is significant. As shown in Table 8, numerous sensitive capabilities are exposed through Init routines that can be indirectly triggered by an unprivileged third-party app, manifesting as privilege escalation vulnerabilities. The vulnerabilities we found pose serious threats to the security and privacy of end users.

In the following, we discuss some representative cases that we have exploited on stock devices. Note that we could only exploit a limited number of findings on live devices due to the lack of physical devices, in our possession, compatible with each impacted firmware in the data set.

Case Study 1: Disabling SELinux. Security-Enhanced Linux (SELinux) is the default security module to manage mandatory access control security policies for all processes on the device. Since Android 5.0, SELinux has been enabled by default, serving as an integral part of the Android security model. We identified a severe vulnerability where an exposed `Init` service can be used to globally disable SELinux enforcement. This impacted six different firmware from the same vendor where one of the seven detected instances can be exploited without user interaction, whereas the other six require a button click. In the affected vendor's firmware, they have included a custom Init service named `wifitest` that, when launched, executes a shell script as the `root` user. The shell script calls `setenforce 0` to disable SELinux, then resets the Wi-Fi interface. Interestingly, in the same `.rc` file where the `wifitest` service is declared, two property triggers have the actions to start this service. One impacted firmware had a privileged app that can be used by attackers to launch the service in the background, without any user interaction. Additionally, five firmware had six privileged apps that could also launch the service upon clicking on a button in their exported GUI.

Case Study 2: Capturing modem and network traffic. Certain Android 9 firmware from two popular Android vendors contained a pre-installed app that utilizes Init services to capture modem and network traffic. On these firmware, third-party apps can send an IPC message to an exported broadcast receiver component of the pre-installed app to start and stop capturing of modem and network traffic on demand. The pre-installed app interacts with multiple custom Init services to capture traffic and store them on external storage. These Init services were all running as the `root` user and used a common shell script where each service passed a different hard-coded string parameter to the shell script to capture data from different interfaces. The script captured traffic to internal storage and then moved the captured traffic to external storage upon completion. These captured records contained significant amount of sensitive data, such as network packets, SMS messages, and phone calls. We were notified by the two impacted vendors that this flaw was introduced by mistake into production builds by a common chipset provider that both vendors had used, and in fact impacted more firmware than in our data set.

Case Study 3: Reading sensitive logs. Three popular Android vendors exposed sensitive system logs via Init services that write the resulting log files on external storage. These system logs provide a timestamped trace of messages, events, and stack traces. Android offers a shared logging mechanism wherein any app can write arbitrary log messages using standard framework APIs. Processes do not always sanitize sensitive user data prior to writing it to the log; therefore, the Android system does not allow third-party apps to access the global system log. Since Init services tend to run as privileged users, they can access sensitive logs from all processes. In each of the three cases, the vendors used an Init service to execute a shell script as the `root` user to execute the `logcat` command. Two of the three vendors also exposed the output of the `dumpsys` command that calls routines in each framework service to dump its state which tends to contain sensitive information. Active monitoring and mining of these logs using regular expressions by an adversarial local process poses a serious risk to user's security and privacy.

Case Study 4: Screen recording. One vendor had five firmware that exposed the capability to initiate a screen recording wherein the resulting recording file is made available to

other processes on external storage. A screen recording provides an actual screen capture and allows an adversary to monitor the contents of device screen and the actions taken by a user. The recordings can reveal data such as passwords, credit card numbers, notification and message content, and other sensitive information. The screen recording was performed in a shell script using the standard `screenrecord` command where the recording has a duration of 30 min.

6.5 Runtime Performance of DEFINIT

We conclude our evaluation by providing measurements of the runtime performance of DEFINIT. We implemented DEFINIT in 7K-SLOCs of Python on top of BinaryNinja [26] for ELF analysis and Kryptowire's internal Android static analysis engine [27] for app analysis. We conducted our analysis on one Ubuntu 20.04 server with 8-core Intel(R) Xeon(R) E5-4620 2.20GHz and 512 GiB of RAM.

DEFINIT took about 5 min on average to unpack a firmware, with 90% of the images finished unpacking in less than 20 min. Processing Init files, collecting traces, and building IDGs took about 30 min on average, with 90% finishing in less than 50 min. Analyzing pre-installed apps took 7 min on average with 90% of the apps finishing in less than 10 min. Each firmware image was analyzed to completion separate from other images and we did not perform any particular optimizations to improve the performance of DEFINIT. Overall, 90% of the firmware finished in less than 70 min end-to-end which is reasonable in practice.

7 Discussion and Future Work

Analysis Limitations. The goal of this study is to explore the impact of Init routines added by vendors to Android and called from privileged apps with potentially lax app component access control. Towards this end, we developed DEFINIT to help us conduct this study. The goal of DEFINIT itself is not to automatically reason whether an identified exposed routine is exploitable or not, but to identify instances that are of potential security impact, bringing them to the surface for an analyst to further investigate and verify. Automatically reasoning about exploitability is an extremely challenging task that has no viable solution in practice [28, 29].

The analysis we performed in §5 is conservative as we tuned our analysis to avoid the constructs that are known to result in false information flows when performing static analysis. These constructs are commonly handled in an unsound manner in practice to avoid overapproximations that may result in too much noise in the findings that analysts have to comb through. For instance, the ICFGs constructed by DEFINIT for ELF binaries and apps were under-approximated to avoid noise in the results as we limited indirect/virtual call resolution to only indirect calls that have one possible candidate callee based on the call receiver information available at an indirect call site. Other constructs that we did not handle include reflection, flows through containers, inter-component communication, and flows that cross between managed and native code (e.g., flows through JNI calls). We also considered permission-protected components as unexported, regardless of the permission protection level [30].

For trace collection, we could have opted for more involved techniques or even firmware emulation [31, 32], though this comes with a multitude of nontrivial challenges beyond the scope of this work [33–35]. From a practical perspective, we believe our analysis was at an adequate level given the findings and goals of this study. More sophisticated analysis can be incorporated in the future to detect obfuscated or deeply-buried behaviors.

Manual Effort. The manual steps performed in this study were pertinent to shortlisting sensitive commands and APIs, developing the detection rules, and analyzing the annotated traces produced from DEFINIT that matched interesting rules. Enumerating and shortlisting the sensitive commands took one day for three persons.

Developing the detection rules used in DEFINIT took about four workdays for one person. We believe our selection provides reasonable coverage for the purpose of this study, though it is straightforward to add more commands and rules in the future as needed. This step is standard in behavioral binary analysis in practice and is unlikely to get fully automated as it requires expert knowledge. It may be possible to automate rule creation to some extent by using data mining techniques [36, 37] on a large labeled corpus of traces of Android-specific potentially sensitive behaviors or a generic model of what constitutes a sensitive behavior on Android. This can be an interesting direction for future work.

Analyzing and verifying the findings in Table 8 took about seven workdays for one person. Since the execution paths identified by DEFINIT cross multiple OS layers, this makes end-to-end automated dynamic verification extremely challenging, which, at a minimum, would requite a rooted target device and an advanced Android-aware, cross-layer dynamic symbolic execution engine. Overall, the manual effort involved was quite reasonable given the number of firmware and apps in our data set and the number of cases we verified.

SELinux and Exploitation. We made the assumption that vendors have configured their firmware images properly for their customizations to work as intended. This includes configuring the necessary SELinux labels, rules, and transitions for their custom routines to function. This also extends to the use of Vendor Init [38] where vendors are expected to place vendor Init `.rc` files and binaries in `/vendor` as needed for them to run under a SELinux domain separate from the system Init domain. DEFINIT detects behaviors that can be exploited through individual pre-installed apps, and all constructs (property names, values, exceutables, commands and APIs) along

a vulnerable path are hardcoded. Therefore, SELinux transitions should not block these flows since the involved actors (pre-installed apps, Init routines and their executables) are the ones expected by SELinux and intended to operate in this manner, unless there are considerable errors on the part of vendors due to a lack of testing. In the cases we dynamically verified, we did not encounter any SELinux restrictions preventing exploitation.

For the scenarios where one sensitive behavior could be split between multiple apps (e.g., attacker invokes one pre-installed app to record a video then a different app to move files to external storage), it may be possible that SELinux prevents exploitation of these behaviors if the triggered routines have different SELinux contexts and the vendor did not add transitions that allow these behaviors to manifest. We leave detecting these multi-app behaviors and handling their SELinux constraints to future work.

Threats to Validity. In our implementation of DEFINIT, we did not check for dynamic access control constructs (e.g., dynamic permission checks, UID checks, confirmation dialogues) that may fall on the path from a pre-installed app to the call site setting a system property. We manually checked only the findings in Table 8 for these constructs during verification. Therefore, the results provided in Tables 7 and A.6 should be taken with this in mind. Reasoning about dynamic access control automatically is a challenging task that requires modeling relevant code constraints dominating a call site setting a system property, modeling runtime environment constraints, and solving these constraints using a symbolic solver, which we leave for future work.

While we tried to cover a representative sample of the Android market, our firmware data set was not uniform across all vendors and Android versions. Some of the vendors in our data set (e.g., Itel) also had significantly smaller firmware images and fewer Init routines compared to others. The unpacking process of some of the proprietary image formats in our data set may have also missed some files and partitions. Therefore, the differences between vendors in our results may not be statistically significant to substantiate differences in the overall security posture of the vendors and should be interpreted carefully in this regard.

Potential Countermeasures. There are various measures that AOSP, Google, and vendors can take to reduce the security impact of Init customizations. The first step is perhaps for Android Init to default child processes spawned from Init to an unprivileged user and SELinux domain (e.g., a `nobody` user). Defaulting to a low-privilege user and domain can confine the impact of exploiting exposed routines and binaries mistakenly leftover by vendors.

Second, given that Google has established a set of requirements as part of the Android Compatibility Definition Document (CDD) [39] that vendors must adhere to in order to brand their devices as Android-compatible, the CDD should enforce strict requirements on vendors to not add privileged custom Init routines that can be programmatically triggered from outside Init itself unless the functionality is key for normal system operation. This can be a mundane process and may not be straightforward to test by the CDD, but it is essential to confine the impact of exploiting potential flaws introduced by Init customizations.

Third, Android can block interaction between unprivileged apps and pre-installed apps that set system properties. In fact, Android can go a step further by blocking interaction between third-party apps and privileged apps by default unless the user explicitly grants a third-party app the permission to interact with a pre-installed app. This step, despite putting the burden on the user, could easily thwart most privilege-escalation attacks from third-party apps trying to parasitize on privileged apps without user consent. Adopting this approach would likely need to be phased in over time in order to not immediately break the current open communication model Android employs among apps co-located on an Android device. In addition, vendors should enforce proper access control at the boundaries of their privileged apps to minimize confused deputy attacks initiated by enterprising third-party apps trying to indirectly trigger sensitive functionality.

Finally, Android SELinux policies could default to preventing executables launched by Init routines from writing to external storage. This could easily block multiple of the flaws identified in our study that capitalize on leaking information to a publicly-readable path on external storage. A better separation of pre-installed apps where the ones that are likely to be interacted with by third-party apps are not allowed to set Init properties or perform sensitive operations may also help here. Some of the most severe cases (e.g., disabling SELinux) should also display a clear warning and ask the user if the action that was initiated programmatically can proceed. Specifically, enforcing user interaction for many of the extensive system logging routines can help to safeguard the user. This is by no means a perfect solution, but if explained clearly, it will allow the user to have greater control of the security of their device.

8 Related Work

Numerous prior works have studied the security issues introduced by Android vendor customizations at different layers of the Android OS. At at the application layer, Woodpecker [2] was among the very first studies to detect capability leakages on Android. It analyzed eight devices and found that 11 out of 13 privileged permissions can leak to unprivileged apps. SEFA [3] analyzed 10 firmware images and found that over 85% of their pre-installed apps were overprivileged. HareHunter [4] discovered thousands of hanging attribute references (Hares) in 97 firmware images, allowing unprivileged

apps to claim access to potentially sensitive functionalities by using attributes hardcoded in pre-installed apps.

More recently, Gamba et al. [40] conducted a comprehensive study of multiple devices and identified several instances of advertising and data collection without user consent. FirmScope [11] performed a large scale static analysis study of pre-installed apps in more than 2000 firmware images from top Android vendors and identified numerous privilege escalation vulnerabilities due to improper access control in pre-installed apps. The authors of FirmScope identified a few number of apps that were able to set arbitrary system properties, which while relevant to our study, they did not assess the impact of setting these properties nor how they may be related to custom Init routines added by vendors. Nevertheless, privilege-escalation flaws in pre-installed apps in general can potentially enable more attack vectors for launching sensitive Init routines, e.g., by exploiting a command execution flaw in a privileged app to directly call an executable launched by a sensitive Init routine running with the `system` UID.

For security issues introduced to the Android framework layer (sometimes referred to as the Android middleware), Tian et. al. [9] analyzed 2,000 firmware images and identified 3,500 AT Commands invokable over USB, multiple of which can perform sensitive functionalities, such as bypassing the screen lock and factory resetting the device. Most of these commands were hardcoded in custom ELF libraries added by vendors to the framework as part of the Radio Interface Layer (RIL) yet a few of them were also introduced by privileged pre-installed apps. ARF [10] analyzed the AOSP framework and identified cases of confused deputies due to inconsistent access checks in framework service components. FANS [41] fuzzed native framework services on six Android 9.0 devices and identified 30 vulnerabilities and thousands of crashes. These studies are complimentary to our work. Studying Init capabilities leaked through vendor customizations to the Android framework itself (e.g., via new framework APIs introduced by vendors) is a possible interesting area for future work.

At the kernel level, ADDICTED [14] analyzed vendor device drivers and found multiple privilege escalation vulnerabilities that allow third-party apps to perform sensitive functionalities without permission by talking to open interfaces in custom device drivers. BootStomp [13] found eight vulnerabilities in the bootloaders used by a number of devices, allowing attackers to potentially compromise the entire chain of trust established at boot time or cause denial of service. BigMac [16] analyzed the SELinux policies on two devices and identified multiple policy inconsistencies that allow unprivileged actors to load kernel modules and communicate with `root` processes.

To the best of our knowledge, none of the prior studies have analyzed vendor customizations of the Android Init process that are visible at the application layer, and the security impact of these changes, which is what we focus on in this study.

9 Conclusion

Android Init routines can provide privileged operation interfaces to privileged system apps that can trigger them by setting system properties. The privileged capabilities of these Init routines can be exposed to unprivileged third-party apps through open interfaces in privileged apps triggering the routines. To understand the prevalence and security impact of exposed Init routines, we designed DEFINIT as a system to help detect Init routines exposed by privileged apps and their behaviors. We studied 259 firmware covering Android 8 to 11 from the top 21 vendors worldwide and identified numerous vulnerabilities that allow unprivileged third-party apps to perform sensitive functionalities, including capturing network traffic, reading system logs, and disabling SELinux, among others. Our findings demonstrate the significance of these changes to Init and the need for rigorous Android regulations to reduce and confine the impact of potential security weaknesses introduced by vendors.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Antonio Bianchi, for their insightful comments on earlier versions of this work. We thank Jinghan Guo for assisting with firmware unpacking and verification. Part of this work was done while Yuede Ji was at George Washington University.

Opinions expressed in this article are those of the authors and do not necessarily reflect the official policy or position of their respective institutions.

References

[1] "Mobile operating system market share worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide/.

[2] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *NDSS*, vol. 14, 2012.

[3] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[4] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare hunting in the wild android: A study on the threat of hanging attribute references," in *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[5] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE Symposium on Security and Privacy (S&P)*.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint

analysis for Android apps," *Acm Sigplan Notices*, vol. 49, no. 6, 2014.

[7] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[8] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in *2018 ACM SIGSAC Conference on Computer and Communications Security*.

[9] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar *et al.*, "Attention spanned: Comprehensive vulnerability analysis of AT commands within the android ecosystem," in *27th USENIX Security Symposium*, 2018.

[10] S. A. Gorski III and W. Enck, "Arf: identifying re-delegation vulnerabilities in android system services," in *12th Conference on Security and Privacy in Wireless and Mobile Networks*, 2019.

[11] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "FirmScope: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in Android firmware," in *29th USENIX Security Symposium*, 2020.

[12] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: from two permissions to complete control of the ui feedback loop," in *2017 IEEE Symposium on Security and Privacy (S&P)*.

[13] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bootstomp: on the security of bootloaders in mobile devices," in *26th USENIX Security Symposium*, 2017.

[14] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in Android device driver customizations," in *2014 IEEE Symposium on Security and Privacy (S&P)*.

[15] H. Zhang, D. She, and Z. Qian, "Android ion hazard: The curse of customizable memory management system," in *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[16] G. Hernandez, D. J. Tian, A. S. Yadav, B. J. Williams, and K. R. Butler, "BigMAC: Fine-grained policy analysis of Android firmware," in *29th USENIX Security Symposium*, 2020.

[17] "Android Init Language," https://android.googlesource.com/platform/system/core/+/master/init/README.md.

[18] F. B. Cohen and D. F. Cohen, *A short course on computer viruses*. John Wiley & Sons, Inc., 1994.

[19] "bash(1) — Linux manual page," https://man7.org/linux/man-pages/man1/bash.1.html.

[20] J. Six, *Application Security for the Android Platform: Processes, Permissions, and Other Safeguards*. " O'Reilly Media, Inc.", 2011.

[21] "Cve-2018-6597," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6597.

[22] "disclosures/getsuperserial.md at master," https://github.com/rednaga/disclosures/blob/master/GetSuperSerial.md.

[23] "Cve-2020-26964," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-26964.

[24] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao, "The misuse of android unix domain sockets and security implications," in *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[25] "Ransomware uses DDoS attacks to force victims to pay," https://www.bleepingcomputer.com/news/security/another-ransomware-now-uses-ddos-attacks-to-force-victims-to-pay/.

[26] "Binary Ninja," https://binary.ninja/.

[27] "Kryptowire," https://kryptowire.com/.

[28] A. Younis, Y. K. Malaiya, and I. Ray, "Assessing vulnerability exploitability risk using software properties," *Software Quality Journal*, vol. 24, no. 1, 2016.

[29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (S&P)*.

[30] "Component permission protection level," https://developer.android.com/guide/topics/manifest/permission-element#plevel.

[31] T. Eisenbarth, R. Koschke, and G. Vogel, "Static trace extraction," in *Ninth Working Conference on Reverse Engineering*. IEEE, 2002.

[32] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium*, 2020.

[33] X. Meng and B. P. Miller, "Binary code is not easy," in *25th International Symposium on Software Testing and Analysis*, 2016.

[34] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*.

[35] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, 2021.

[36] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 163–182.

[37] E. Raff, R. Zak, G. Lopez Munoz, W. Fleming, H. S. Anderson, B. Filar *et al.*, "Automatic yara rule generation using biclustering," in *13th ACM Workshop on Artificial Intelligence and Security*, 2020.

[38] "Vendor Init | Android Open Source Project," https://source.android.com/security/selinux/vendor-init.

[39] "Android Compatibility Definition Document," https://source.android.com/compatibility/cdd.

[40] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE Symposium on Security and Privacy (S&P)*.

[41] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing Android native system services via automated interface analysis," in *29th USENIX Security Symposium*, 2020.

[42] "Android phone manufacturer market share," https://www.appbrain.com/stats/top-manufacturers.

A Appendix

A.1 Analyzed Firmware

Table A.1 provides the details of the firmware images analyzed by DEFINIT. At the time of writing, Android v11.0 was recently released and only a few vendors provided public Android v11.0 images.

Table A.1: Summary of the tested Android firmware.

Vendor	#Firmware	#Apps	v8	v9	v10	v11
Alcatel	6	1,088	4	2	0	0
ASUS	15	4,093	5	5	5	0
BLU	13	2,034	5	6	2	0
Coolpad	4	609	4	0	0	0
Google	20	3,885	5	6	5	4
Hisense	10	1,489	10	0	0	0
HTC	5	1,148	5	0	0	0
Huawei	6	574	6	0	0	0
Infinix	13	2,896	2	5	6	0
Itel	9	1,529	4	5	0	0
Lava	15	2,579	10	5	0	0
Lenovo	5	774	0	4	1	0
Nokia	11	2,578	3	3	5	0
OnePlus	19	6,450	5	5	5	4
Oppo	19	5,792	5	5	6	3
Realme	14	4,652	0	3	11	0
Samsung	22	9,700	4	4	10	4
Tecno	15	3,142	4	5	5	1
Vivo	9	2,167	4	1	4	0
Xiaomi	19	5,293	5	6	8	0
ZTE	10	2,160	3	5	2	0
total	259	64,632	93	75	75	16

A.2 Firmware Acquisition

We downloaded firmware images from the following sources:

Alcatel	https://alcatelfirmware.com
ASUS	https://www.asus.com/support/
Google	https://developers.google.com/android/images/
Samsung	https://www.sammobile.com
HTC	https://www.htc.com/us/support/
Huawei	https://huaweistockrom.com/
OnePlus	https://www.oneplus.com/support/
Oppo	https://oppo-au.custhelp.com
Vivo	https://vivofirmware.com
Xiaomi	https://c.mi.com/global/miuidownload/index
ZTE	https://www.ztedevices.com/en/support/
Other	https://androidmtk.com
	https://www.stockrom.net
	https://firmwarecare.com
	https://firmwarefile.com
	https://easy-firmware.com

A.3 Market Share of Impacted Vendors

Table A.2 shows the global market shares of the impacted vendors (anonymized) and their shares of verified vulnerabilities in our findings. Note that these are market shares of the vendors rather than the specific impacted devices and Android versions, which we were unable to obtain. We used Android vendor market share data from AppBrain [42].

Table A.2: Anonymized vendor data showing global market share and their ratio of introduced vulnerabilities.

Vendor	%Vulnerability	%Market Share
A	41%	8.9%
B	18%	2.8%
C	16%	7.6%
D	6.3%	0.5%
E	4.9%	0.5%
F	4.5%	0.6%
G	3.9%	10.8%
H	2.4%	0.6%
I	1%	<0.4%
J	1%	<0.4%
K	1%	<0.4%

A.4 Rule Samples

In the following, we present some of the rules used by DEFINIT. Syntactic details have been abstracted and simplified for the sake of presentation.[6]

Disable SELinux One of the rules with the greatest impact on the overall security of the system is disabling SELinux enforcement. Below is a rule that detects instances of executing a command to disable SELinux.

```
conditions:
    setenforce 0
    | setenforce permissive
    | echo 0 > /sys/fs/selinux/enforce
```

[6] Additional rules are available at https://kryptowire.com/definit.

Sniff Modem Traffic The following rule detects calls to the diag_mdlog utility to capture and dump modem traffic to a path on external storage, which is readable by any app that has been granted permission to read external storage.

```
conditions:
    (diag_mdlog|diag_mdlog_system|oppo_diag_mdlog) * (-f|-o
        ) $sdcard/*
```

Here, $sdcard is a common internal rule that matches a path prefix on external storage:

```
sdcard: /sdcard | /mnt/sdcard | /storage/self/primary
        | /storage/emulated/0 | /data/media/0
```

Read System Logs The following rule captures the leakage of Logcat logs to external storage.

```
conditions:
    logcat * (-f|>) $sdcard/*
    | logcat * (-f|>) *
      (mv|cp) * $sdcard/*
```

Record Screen The following rule detects the usage of the screen record command where the resulting video file is stored on external storage.

```
conditions:
    screenrecord * $sdcard
    | screenrecord *
      (mv|cp) * $sdcard
```

Factory Reset The following rule detects the sending of a broadcast Intent message that initiates a factory reset of the device, wiping all user data.

```
conditions:
    am broadcast * -a android.intent.action.MASTER_CLEAR *
```

Read Kernel Logs The following rule detects access to the the kernel logs when leaked to a path on external storage.

```
conditions:
    (dmesg|klogd|/proc/kmsg|/dev/kmsg) * (-o|>|-f) $sdcard
    | (dmesg|klogd|/proc/kmsg|/dev/kmsg) *
      (mv|cp) * $sdcard
```

Sniff Network Traffic The following rule detects calls to the tcpdump utility to capture and dump network traffic to a path on external storage.

```
conditions:
    tcpdump * (-w|>) $sdcard/*
    | tcpdump (-w|>) *
      (mv|cp) * $sdcard
```

Read Wi-Fi Passwords The following rule detects access to the contents of the /data/misc/wifi/wpa_supplicant.conf file containing the Wi-Fi passwords which are subsequently written to external storage.

```
conditions:
    cp /data/misc/wifi/wpa_supplicant.conf $sdcard
    | cat /data/misc/wifi/wpa_supplicant.conf > $sdcard
```

A.5 Routines Exposed via the GUI

We provide measurements of exposed Init routines and manually verified vulnerabilities that were only reachable via GUI entry points in Tables A.3 to A.5. Attackers may be able to exploit some exposed routines by tricking the user into interacting with the GUI of an exported component in a privileged app. While this requires user interaction, it is still a valid attack vector with a relatively low complexity [12].

Table A.3: Exposed routines only reachable via the GUI.

Version	Exposed	Exposed Sensitive
8	221 (0;12;2.4)	42 (0;5;0.5)
9	109 (0;5;1.5)	44 (0;5;0.6)
10	78 (0;4;1.0)	35 (0;3;0.5)
11	27 (0;4;1.7)	9 (0;4;0.6)
Total	435 (0;12;1.7)	130 (9;5;0.5)

Counts are 'total (min;max;avg per firmware per version)'

Table A.4: Exposed functionalities only reachable via the GUI.

Category	Total	Apps	Firmware	Vendors
Device settings	37	31	21	8
Sensitive data	8	8	5	2
Networking	32	32	32	9
Package management	29	29	17	3
Permission control	6	6	5	1
Power management	23	17	14	7
Process management	54	54	51	12
UI interaction	0	0	0	0
Total (unique)	103	89	71	14

Table A.5: Verified vulnerabilities requiring user interaction and the functionalities they allow an unprivileged attack app to perform.

Impact	Total	Apps	Firmware	Vendors	Versions
Read system logs	11	11	11	3	9,10
Sniff modem traffic	13	13	13	3	9,10,11
Sniff Wi-Fi traffic	7	7	7	2	10
Sniff Bluetooth traffic	10	10	10	2	10,11
Read dumpstate	10	10	10	2	10,11
Read dumpsys	10	10	10	2	10,11
Read kernel logs	8	8	8	2	10
Read bugreport	8	8	8	2	10
Read radio logs	10	10	10	2	10,11
Disable SELinux	6	6	5	1	8,9
Reboot into recovery	7	7	4	1	8,9
Reboot device	17	17	14	7	8,9,10
Disable Wi-Fi	15	15	8	2	9,10
Disable NFC	2	2	2	1	11
Total (unique)	134	52	35	9	8–11

A.6 Commands Called by Exposed Routines

Table A.6 shows the breakdown of sensitive commands called by the exposed Init routines identified by DEFINIT.

Table A.6: Init routines calling sensitive commands/APIs, the number of apps exposing them, and the impacted firmware. Multiple matches for the same command/API are counted only once per unique routine.

Category	Command	Total	Direct	Indirect	Apps	Firmware	Vendors
Device settings	hid	0	0	0	0	0	0
Device settings	settings	0	0	0	0	0	0
Device settings	locksettings	0	0	0	0	0	0
Device settings	svc	37	15	22	31	21	8
Device settings	ime	13	13	0	13	13	2
Sensitive data	atrace	76	76	0	46	24	2
Sensitive data	bugreport	19	16	3	19	19	3
Sensitive data	content	0	0	0	0	0	0
Sensitive data	diag_klog	0	0	0	0	0	0
Sensitive data	diag_mdlog	20	1	19	7	7	3
Sensitive data	diag_socket_log	5	0	5	5	5	2
Sensitive data	diag_uart_log	0	0	0	0	0	0
Sensitive data	dumpstate	39	39	0	39	39	3
Sensitive data	dumpsys	103	91	12	62	36	7
Sensitive data	logcat	81	65	16	48	44	7
Sensitive data	ramdump	30	30	0	30	30	3
Sensitive data	record_stream_new	11	6	5	7	6	2
Sensitive data	screencap	0	0	0	0	0	0
Sensitive data	screenrecord	6	6	0	6	6	1
Sensitive data	tcpdump	32	15	17	19	19	4
Networking	dnsmasq	14	14	0	14	14	1
Networking	ifconfig	4	0	4	3	3	1
Networking	iptables	0	0	0	0	0	0
Networking	telecom	0	0	0	0	0	0
Networking	send	4	2	2	4	4	2
Networking	sendfile	0	0	0	0	0	0
Networking	sendfile64	0	0	0	0	0	0
Networking	socket_local_server_bind	75	29	46	75	74	12
Package management	applypatch	0	0	0	0	0	0
Package management	pm	3	0	3	3	3	2
Package management	dpm	0	0	0	0	0	0
Package management	insmod	30	0	30	30	18	3
Package management	patchoat	0	0	0	0	0	0
Permission control	keystore	0	0	0	0	0	0
Permission control	appops	8	8	0	8	8	2
Permission control	setsid	0	0	0	0	0	0
Permission control	load_policy	0	0	0	0	0	0
Permission control	setenforce	7	0	7	7	6	1
Power management	thermal_engine	6	0	6	3	3	1
Power management	__reboot	11	11	0	11	11	6
Power management	android_reboot	13	4	9	13	10	2
Power management	reboot	26	16	10	22	18	8
Process management	cmd	2	0	2	1	1	1
Process management	killall	0	0	0	0	0	0
Process management	killpg	0	0	0	0	0	0
Process management	ptrace	2	0	2	1	1	1
Process management	service	80	76	4	80	69	12
UI interaction	virtual_touchpad	0	0	0	0	0	0
UI interaction	am	34	17	17	34	31	5
UI interaction	input	7	0	7	7	7	3
UI interaction	sendevent	0	0	0	0	0	0
UI interaction	monkey	12	12	0	12	12	2
UI interaction	uiautomator	0	0	0	0	0	0