Cross-vendor Security Analysis of Android Unix Domain Sockets

Mounir Elgharabawy

A Thesis
in
The Concordia Institute
for
Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Master of Applied Science (Information and Systems Engineering) at
Concordia University
Montréal, Québec, Canada

September 2021

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mounir Elgharabawy**

Entitled: **Cross-vendor Security Analysis of Android Unix Domain Sockets**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information and Systems Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Jamal Bentahar

_____ Examiner
Dr. Ivan Pustogarov

_____ Examiner
Dr. Jamal Bentahar

_____ Thesis Supervisor
Dr. Mohammad Mannan

_____ Thesis Supervisor
Dr. Amr Youssef

Approved by _____
Dr. GPD, Graduate Program Director

October 5, 2021 _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Cross-vendor Security Analysis of Android Unix Domain Sockets

Mounir Elgharabawy

The Android operating system is currently the most popular mobile operating system in the world. Android is based on Linux and therefore inherits its features including its Inter-Process Communication (IPC) mechanisms. These mechanisms are used by processes to communicate with one another and are extensively used in Android. Although the Android-specific IPC mechanisms have been studied extensively, Unix domain sockets have not been studied as much despite playing a crucial role in the IPC of highly privileged system daemons. In this thesis, we propose SAUSAGE, an efficient novel static analysis framework to study the security properties of these sockets. SAUSAGE considers access control policies implemented in the Android security model as well as authentication checks implemented by the daemon binaries. It is a fully static large-scale analysis framework specifically designed to analyze Unix domain socket usage in Android system daemons. We use this framework to analyze 200 Android images across eight popular smartphone vendors spanning Android versions 7-9. As a result, we uncover multiple access control misconfigurations and insecure authentication checks introduced by vendor customization. Our notable findings include a permission bypass in highly privileged Qualcomm system daemons and a vendor-specific daemon exposing an unprotected socket that allows an untrusted app to set the scheduling priority of other processes running on the system.

Acknowledgments

I would like to thank Dr. Amr Youssef and Dr. Mohammad Mannan for their support and guidance throughout my Masters studies. Their input has been invaluable to guide my efforts to fruition, especially at times of uncertainty. Without their advice and support, throughout this project and earlier ones, this work would not have come to life. I would also like to express my gratitude to our collaborators, Dr. Kevin Butler and Dr. Byron Williams of the University of Florida, for providing their expert knowledge in the field of Android OS security and for steering our research to its current state. I would like to give thanks to Blas Kojusner, my colleague in the University of Florida, who collaborated with me for the whole project. I am grateful to Grant Hernandez, the primary author of BigMAC, for spending time with us in order to solve technical issues with our project.

Finally, I would like to express my gratitude to my friends and family for their support during the hardest times of my Masters studies, especially the ones who sourced and donated their Android phones for the ground truth evaluation and proof-of-concept exploit development.

Contents

List of Figures

List of Tables

List of Acronyms

AOSP Android Open-Source Project.

AVrules Access Vector rules.

CDD Compatibility Definition Document.

CFG Control Flow Graph.

DAC Discretionary Access Control.

DNS Domain Name System.

GID Group ID.

IPC Inter-Process Communication.

MAC Mandatory Access Control.

MSM Mobile Station on Modem.

OS Operating System.

PID Process ID.

PoC Proof of Concept.

QTEE Qualcomm Trusted Execution Environment.

SELinux Security-Enhanced Linux.

SoC System on Chip.

UID User ID.

Chapter 1

Introduction

In this chapter, we introduce the reader to our problem area, by giving a brief overview of Android and Unix domain socket usage. We then motivate our work, which is detailed in this thesis, by providing examples of past vulnerabilities that arose due to the insufficient protection of these sockets. We also point out the gap left by past research which we successfully address in our work. Afterwards, we propose our analysis tool, SAUSAGE, which performs **S**ecurity **A**nalysis of **U**nix domain **S**ocket us**age** in Android. This is followed by our most notable contributions and findings, and our responsible disclosure process. In the final section, we provide an outline of the organization of the rest of the thesis.

1.1 Overview

Android is currently the most widely used Operating System (OS) in the world, occupying around 43% of the OS global market share [8] and 73% of the mobile OS global market share [7]. Android is developed and maintained by Google and is originally based on Linux, inheriting many of its features. It is open-sourced under the Android Open-Source Project (AOSP). One of the fundamental features any modern operating system provides is Inter-Process Communication (IPC) mechanisms, a method of communication between

processes to allow for more functionality on a system. Android is no different; it provides a variety of Android-specific IPC mechanisms (e.g., Binder, Intents, Messenger) and also inherits the traditional IPC mechanisms available in a Linux environment. Of these traditional Linux IPCs are Unix domain sockets. In Android, Unix domain sockets are primarily used by native system services or daemons, processes that constantly run in the background, as one of the main methods of communication with the Android Framework layer. These daemons handle critical low-level tasks, such as performing Domain Name System (DNS) queries or handling cellular communication, making them high-valued targets for exploitation. Like any IPC, these sockets present potential attack vectors for confused deputy-like attacks when used by a highly privileged process, and generally increase the process's attack surface. This is due to the fact that, aside from Android system services and apps, third-party apps with the proper permissions can access Unix domain sockets.

1.2 Motivation

Multiple vulnerabilities and exploits arose due to the misuse of these sockets. One example is CVE-2011-3918 [36] where an unprotected socket to the Zygote process could be leveraged to perform a denial of service attack on a device running AOSP Android 4.0.3. Another example is the "HTC WeakSauce" exploit which makes use of a socket connection to the privileged *dmagent* system daemon to achieve privilege escalation to root [28]. CVE-2013-4777 and CVE-2013-5933 [37, 38] are privilege escalation vulnerabilities affecting Motorola devices, the root cause of which was an unprotected socket to the *init* process. More recently, an information disclosure vulnerability was discovered on Huawei phones which allows attackers to gather screenshots and kernel and system logs [26]. The vulnerability is exploited by sending commands via an exposed socket to a vendor-customized version of the *debuggerd* daemon. These vulnerabilities indicate that unprotected sockets can degrade the security of the system whether they originate in AOSP Android or in

vendor customization. Unfortunately, previous research in the Android IPC domain has mainly revolved around Android-specific IPCs such as Binder and Intents, leaving much to be desired in the evaluation of traditional Linux IPCs such as Unix domain sockets.

This warrants a closer look to measure the prevalence and severity of this issue in Android systems. Shao et al. [46] carried out the first study examining the misuse of Unix domain sockets in Android. Their results indicate that inadequate protection of these sockets is a common pitfall in both Android apps and system daemons. Their approach succeeded at statically analyzing Android apps at scale, but fell short in the case of system daemons which are arguably much more valuable targets for exploitation. This shortcoming is caused by their adoption of a dynamic analysis approach to avoid challenges such as reasoning about the complex interaction of Android access control layers, extracting firmware images from different vendors with different formats, and statically analyzing system daemon ARM binaries accurately. As a consequence, their analysis requires access to a running, rooted Android device, and thus only covers two vendors across three Android versions.

1.3 Proposed Solution

To address the absence of a cross-vendor large-scale analysis tool for Unix domain sockets in the literature, we propose SAUSAGE, a large-scale static analysis framework to identify valid socket connections that untrusted apps can establish to system daemons on an Android device, without the need of a running device. Given an Android firmware image, our framework analyzes access control policies and performs static binary analysis on daemon binaries to discover socket addresses that an untrusted app can connect to and any authentication checks implemented in the binary. We overcome the challenge of reasoning about Android access control policies by using a modified version of BIGMAC, a tool originally developed by Hernandez et al. [25], and implement our own binary analysis component.

The framework extracts the system's SELinux policy, system daemon binaries and init RC files from an Android firmware image. It analyzes the SELinux policy to determine which system daemons an untrusted app can communicate with. By using inter-procedural data-flow analysis, it then detects socket addresses, their access control credentials, and any authentication checks in the system daemon binaries with high accuracy. SAUSAGE can do this on a large scale as it fully analyzes a firmware image in around 13 minutes.

We used our framework to analyze 200 Android firmware images spanning eight different vendors and Android versions 7-9. The results of our analysis are worrisome; all vendors except AOSP had access control issues that allow an untrusted app to communicate to highly privileged daemons. These include HTC dmagent which has been previously exploited in "HTC WeakSauce," and Samsung's Professional Audio service which in allows any app to set its process scheduling priority. We also identify insecure authentication practices used by these daemons, such as checks based on an app's process name, which can be trivially spoofed. Additionally, we demonstrate that our approach can uncover IPC sockets that would have been missed by the dynamic analysis approach used by Shao et al [46].

1.4 Contributions and Notable Findings

1.4.1 Contributions

1. We propose an access control-aware, fully static, large-scale framework to analyse Unix domain socket usage in Android system daemons.

2. We use a novel methodology based on a modified version of BIGMAC to reason about Android's complex interaction of access control layers through static analysis.[1]

[1]We will publish the source of the binary analysis module to the community, as well as the modified version of BIGMACwe use as part of our framework.

3. Using this framework, we conduct an analysis of a dataset[2] of 200 Android factory images spanning versions 7.0-9.0 across 8 different vendors, including prominent players such as Samsung and Xiaomi, which account for over a third of the mobile vendor market share worldwide [48], and others such as Asus, Motorola, HTC, Huawei, Asus and AOSP, to find system daemon sockets accessible to untrusted apps. We use the term untrusted app to reference third-party applications a user can install. We also compare our results to the ground truth from three running devices and find that our framework achieves 100% accuracy in detecting socket addresses and their MAC and DAC credentials.

4. We find multiple instances of unprotected Unix domain sockets to root processes that could lead to exploits such as HTC WeakSauce [28].

5. Our approach detects Unix domain sockets that are created under certain conditions and would not have been detected through past approaches relying dynamic analysis alone.

Other works conducted during the tenure of this thesis were published in [10, 11, 12].

1.4.2 Notable Findings

1. We found two highly privileged Qualcomm system daemons, *cnd* and *dpmd*, where a faulty authentication mechanism is used to authenticate the peer, relying only on its process name. However, the process name can be easily spoofed in both cases. Through static analysis, we infer that this allows clients to get/set network settings such as WiFi AP, WiFi P2P, and Default Network settings, by sending the appropriate command over the cnd socket.

[2]The framework itself is compatible with Android versions 7.0 and above, and is vendor-independent. However, our evaluation dataset is limited by the firmware unpacking tool being used.

2. In 25 Samsung Android 7.0-7.1 images, we found that the daemon *apaservice* listens over a socket that can be used to request changing the scheduling priority for any process to any priority. This can result in DoS of the Samsung audio subsystem, as well as potentially arbitrary processes running on the device. Additionally, the daemon is vulnerable to stack overflow, potentially leading to privilege escalation.

3. There are multiple instances of overly permissive SELinux policies in five of the eight vendors we analyzed. These policies allow socket communication between untrusted apps and highly privileged system daemons, weakening the system's overall security posture. Examples of these daemons include *dumpstate* and *rild* in HTC, and *dpmd* in most vendors.

4. We discovered multiple instances of vendor customization of AOSP binaries that expose additional unprotected sockets. One example is HTC *rild* where two custom sockets were added that are configured to be accessible to an untrusted app.

We have also contacted the affected vendors as part of our responsible disclosure process. Samsung acknowledged the vulnerability in *apaservice* and is currently working on a patch. They also rewarded our findings through bug bounty program on Bugcrowd. Qualcomm responded that the unprotected *cnd* socket is deprecated starting Android 8.0. Thus, they will not be patching the affected systems, despite around 180 million users relying on the affected Android versions (7.0-7.1) [6].

1.5 Outline

The rest of the thesis is organized as follows: Chapter 2 provides an overview of topics relevant to our work, discussing the Android security model, BigMAC (the access control policy analysis tool we use), native Android system daemons, Unix domain sockets, our

analysis threat model, and a review of studies related to our project. In Chapter 3, we describe the design and implementation details of our tool, SAUSAGE. Chapter 4 details our findings and their security impact, and presents two case studies of vulnerabilities based on our findings. In Chapter 5, we evaluate SAUSAGE's performance and correctness against the ground truth from real world Android device, and discuss our tool's limitations. Finally, in Chapter 6, we conclude our thesis and discuss future directions to improve our work.

Chapter 2

Background, Threat Model and Related Work

In this chapter, we start by giving an overview of the different layers of the Android Security model. Next, we discuss BIGMAC, a fine-grained SELinux policy analysis tool for Android that serves as a core component in our analysis pipeline. This is followed by a discussion of system daemons, and where they fit in the Android OS architecture. We then examine the different types of Unix domain sockets available on Android and their properties. Following, we define our study's threat model. Finally, we review related work spanning two broad research areas in Android security.

2.1 Background

2.1.1 Android Security Model

The Android security model implements various layers. Apps run in sandboxes defined by the creation of a unique Linux User ID for each application at install time. Processes can only communicate with one another via enforced Mandatory Access Control (MAC). This

is a feature implemented in Security-Enhanced Linux (SELinux). Interested readers can find a comprehensive discussion of the Android security model in [33]. In this section, we explain the key concepts behind it that are relevant to our work.

Discretionary Access Control on Android

DAC is an access control model that is used by Linux. It is implemented in Android by using a fixed set of user IDs (UID) and group IDs (GID) for system related purposes and limiting a range of user IDs for dynamically installed applications. Android limits the number of processes that can run as root, therefore a high privileged process would typically run under the system UID, or another UID specific to execute the role intended for the process (e.g. radio, bluetooth). This avoids granting more permissions than necessary to a process which can inherently avoid security issues. Untrusted apps are assigned a unique UID from a specified range of IDs that are available. This prevents the third-party applications from having more access than necessary on any other files that are not included in their installation.

SELinux

This is a set of kernel modifications and user-space tools that have been added to various Linux distributions. It was introduced into the Android platform in 2013. SELinux contains a set of rules that are based on file labels which contain information such as user, role, type, and level. These rules determine what types and actions a process has access to and are structured to group items together based on their accessibility. SELinux implements MAC as a security module that uses the Linux Security Module framework. In Android, SELinux is not only restricted to access control for files, but has been extended to manage access control for IPC mechanisms such as Binder and Unix domain sockets. Thus, for processes to communicate with one another, the communication must be explicitly allowed by the

SELinux policy.

SELinux Policies are created by combining the core AOSP policy with device-specific customization. Policies are a set of rules that guide the SELinux security engine. In it there are types for file objects and domains for processes. The SELinux Policy uses roles to limit the domains that can be accessed and has user identities to specify the roles that users can have. New rules can be added into the SELinux Policy which is then preprocessed and built into the policy.conf file. In this model, all third-party applications are assigned the same `untrusted_app` SELinux domain.

The SELinux Policy is variable between different Android vendors and versions. Android vendors often customize this policy to facilitate inter-operability between vendor-specific components and apps, and pre-existing AOSP components, often to the detriment of the system's security posture [40, 25].

Supplementary Groups

Adding a supplementary group ID to an application will grant it all the privileges of the specified group. The groups for an application are assigned within the manifest file. An example of some supplementary groups would be the Bluetooth group or the Internet group. The permissions of a supplementary group can be enforced at the kernel level or at the Android Framework level depending on the functionality granted to the group.

Middleware Permissions

The Android middleware layer contains a reference monitor that mediates inter-component communication [20]. Middleware permissions grant apps access to resources and services that are provided by the Android operating system rather than the Linux kernel. Some of these middleware permissions map to supplementary groups. Thus, an app with one of these permissions has the same privilege as a process with the corresponding group ID.

2.1.2 BigMAC

BIGMAC [25] is a fine-grained SEPolicy static analysis tool. In this section we will discuss the process behind how BIGMAC works and what results can be generated. BIGMAC first walks the filesystem of the unpacked image and extracts the files' DAC file permissions, SELinux labels and Linux capabilities. Then, it parses the system's init scripts and simulates commands that affect the filesystem (e.g., `mkdir`, `chmod`, etc.) as well as `service` commands which execute service binaries. Performing boot emulation is required to create files in the `/sys`, `/dev` and `/data` directories, which would not be present in a static firmware image.

After the boot process is emulated, BIGMAC begins the Backing File Recovery step, where it assigns the appropriate SELinux labels to all files in the extracted filesystem.This is done by decompiling the extracted binary SEPolicy file to a multi-edge directional graph via the Access Vector rules (AVrules). Afterwards, it correlates policy types to actual files on the initialized filesystem. File objects are straightforward to correlate since their SELinux labels are captured in the extraction step. For process subjects, Type Enforcement rules related to process transitions are inverted and processed allowing for the correlation of subject types and their executable binary backing files.

Using the full set of subject nodes, BIGMAC constructs a dataflow graph which simplifies the SELinux policy's access vectors into a read/write model. The dataflow graph captures all data flows allowed by AVRules for all subjects and objects by considering vectors that imply a read or a write. In the dataflow graph, objects can be one of two types: file objects and IPC objects. As discussed in the previous step, file objects can contain multiple backing files, each with its own MAC/DAC/CAP metadata. On the other hand, IPC objects typically do not have any backing files and are tagged with the underlying AVClass. For instance, all classes that derive from the `socket` class are tagged as IPC objects.

The recovered subject nodes are also used in the Process Inflation step. For each subject

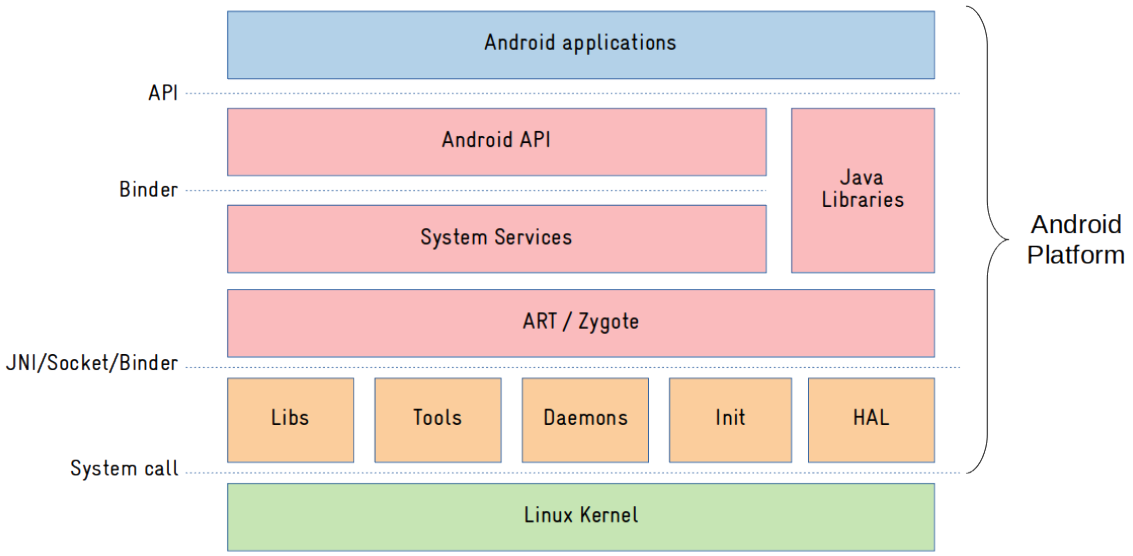

Figure 1: Android architecture [41]

node, BIGMAC attempts to match the subject node to a service definition by comparing the subject's backing file with the binary file in the service definition. If the service is enabled and is not a one shot (transient) process, the service's defined security options are assigned to a new process. This process is then inserted into a concrete process tree.

The final step in the process is the attack graph instantiation. In this step, all file objects within the dataflow graph are expanded, such that each file corresponds to one node in the graph, encompassing all of this file's MAC/DAC/CAP attributes. All of the edges to and from the original file object are duplicated for each individual node. This expanded graph is overlaid onto the concrete process tree, whereby, for each process in the process tree, all in- and out-edges in the corresponding subject in the dataflow graph are copied to the process tree. In the resultant graph, concrete processes have concrete edges to all the objects they can read from or write to. BIGMAC then uses this graph to generate Prolog facts that can be used for dataflow paths in that graph.

2.1.3 System Daemons

A daemon is defined as a process that runs in the background without owning a GUI. In a Unix environment, daemons are often started by the *init* process. Android is no different. In Android, *init* starts daemons specified by service definitions in init RC files. Instead of each daemon being started with the same credentials as *init* (root), each service definition specifies the daemon's desired DAC credentials. Figure 1 shows where these daemons are positioned with respect to the entire Android architecture. Unmodified AOSP Android contains daemons which provide services essential to the Android framework, such as logging, DNS resolution, profiling and others. We refer to these daemons as "AOSP Daemons." AOSP daemons are available in all Android systems as part of Vanilla Android. The AOSP version of these daemons' source code is made available by AOSP, although vendors might make proprietary customizations to them in their own distributions of Android. These daemons often purposefully expose sockets for communication with an untrusted app to implement various functionalities. Additionally, hardware vendors, such as Qualcomm, implement daemons to act as an interface to hardware peripherals. Custom daemons can also be added by vendors, such as Samsung, to their custom Android distributions. We term these daemons "Vendor-specific Daemons."

As an example, netd is an AOSP daemon which controls network interfaces, their configuration, and other network-related functionality [31]. This daemon typically exposes four sockets in the RESERVED namespace: netd, dnsproxyd, mdns and fwmarkd. Of these sockets, only dnsproxyd and fwmarkd are accessible to untrusted apps with the INTERNET permission. The dnsproxyd socket is used by all apps to perform DNS name resolution, a crucial step need to contact a server by translating its domain name to an IP address. In fact, this is the underlying mechanism for any process to perform DNS name resolution in Android, as it is implemented in Android's version of libc, Bionic. In Samsung Android 7, the AOSP netd daemon is modified, and a new RESERVED socket is added to netd

with address napproxyd. This addition is part of Samsung's Knox framework and allows recording of the domain associated with network activities for the Knox NetworkAnalytics feature. Indeed, Samsung's version of bionic hooks the `connect`, `recvfrom` and `accept` functions to send this command before proceeding if the net.knox.nap system property is set.

2.1.4 Unix Domain Sockets

A Unix domain socket is a communications endpoint for exchanging data between processes on the same host operating system. It can also be referred to as an inter-process communication socket. The main difference between Unix domain sockets and Internet sockets is that a Unix domain socket is an IPC where all communication occurs strictly within the operating system kernel. Internet sockets use an underlying network protocol for communication. Unix domain sockets also have what is called a namespace, or a unique identifier to an object of a certain kind, that is used to label the socket types. There are three types of Unix domain socket namespaces in Android, as can be seen in Table 1.

Namespace	**SELinux Enforcement**		**DAC Enforcement**
	IPC	File Contexts	File Permissions
RESERVED	✓	✓	✓
FILESYSTEM	✓	✓	✓
ABSTRACT	✓	✗	✗

Table 1: Security enforcement corresponding to Android Unix domain socket namespaces

FILESYSTEM. Sockets with this namespace are associated with a file on the filesystem and are created by a processes that needs them. Once a socket file is created it will be protected by the Discretionary Access Control (DAC), as well as the mandatory access control, or the MAC. Only processes with the proper MAC and DAC credentials can communicate with these socket files.

RESERVED. This namespace is introduced in Android and falls under the FILESYSTEM

namespace and thus inherits its access control properties. These socket files are created by init and are located under `/dev/socket`. The name indicates that these socket files are reserved for system use. The socket file descriptors are made available to their owner service daemon through an environment variable named `ANDROID_SOCKET_<addr>` where `<addr>` is the address of the socket in the RESERVED namespace.

ABSTRACT. These sockets allow a program to bind a Unix domain socket to a name without the name being created in the filesystem. The socket's name begins with a null byte which removes the need to create a filesystem path name for the socket.

There are three prerequisites for a process to be allowed to establish a connection to a FILESYSTEM (or RESERVED) socket. First, the connecting process must be allowed to communicate to the server process through the Unix domain socket IPC by SELinux. Second, the connecting process must be allowed to write to the socket file, based on its file context in SELinux. Third, the connecting process must have the appropriate UID or GID to write to the socket file, depending on the socket file's DAC file permissions. On the other hand, only the first prerequisite is needed in the case of ABSTRACT sockets since file-based access control policies are not applicable to them. As a result, ABSTRACT sockets are the least secure of the three namespaces.

Furthermore, Unix domain sockets can only be bound by one process. Filesystem MAC and DAC can restrict the creation of sockets under certain directories to a set of processes, preventing untrusted apps from binding sockets used by system daemons. This does not apply to ABSTRACT sockets, however, allowing a malicious app to DoS a system daemon by occupying an ABSTRACT address if the app manages to bind the socket address before the daemon. Daemons started by init at boot-time are always started before apps. However, if at any point the daemon closes the socket bound to an ABSTRACT address or the daemon itself restarts, a malicious app can bind that address and prevent the system daemon from re-binding it, causing DoS of that daemon. Even further, the app can spoof the daemon and

potentially receive privileged information through the Unix domain socket. This can only happen if the sender is allowed to connect to untrusted apps by the SELinux policy.

2.2 Threat Model

Unix domain sockets can only be accessed from processes that have proper permissions when checked by the MAC and DAC. Our threat model specifically focuses on untrusted apps, labeled `untrusted_app` in SELinux. In our threat model, an app is allowed to obtain middleware permissions that can be granted to any `untrusted_app`. Middleware permissions are not part of the MAC or DAC, so they are not directly considered when a socket is being accessed. However, the DAC supplementary groups assigned to a untrusted app depend on the permissions it has.

There are four permissions that any untrusted app can request which map to supplementary groups. The `android.permission.INTERNET` permission, which corresponds to the `inet` group, allows the `untrusted_app` to perform network operations, such as opening network sockets. The `android.permission.BLUETOOTH_ADMIN` permission, which corresponds to `net_bt_admin`, allows applications to discover and pair Bluetooth devices. Similarly, the Bluetooth permission labeled `android.permission.BLUETOOTH` permission, corresponding to `net_bt`, allows applications to connect to paired Bluetooth devices. The final permission is the external storage permission, `android.permission.MANAGE_EXTERNAL_STORAGE`, which belongs to the `external_storage` group, allows an application a broad access to external storage in Scoped storage, a feature in Android allowing an application to only have access to their application directory on external storage plus any media created by the app [5].

2.3 Related Work

In this section, we outline related studies to our research. We start by exploring past research on Android IPC mechanisms, zooming in on the first and only study to examine Unix domain socket usage in Android. In section 2.3.3, we review research on Android access control policy analysis, since it plays a significant role in protecting IPC in Android. Our work significantly relies on Android access control policy analysis and Android IPC security analysis, and bridges both of these research areas together. By analyzing Android MAC and DAC, we narrow down the attack surface exposed to our chosen threat model, and exclusively analyze system services and IPC mechanisms that constitute this attack surface. Since all Android IPC is governed by MAC (and sometimes by DAC), we believe that our approach may serve as a good base for future work examining the security of Android IPC mechanisms (especially statically).

2.3.1 Android IPC

Android IPC security has long been the focus of a large body of research. Most of this research, however, centered on either the Binder IPC interface [22, 29, 53, 17, 52, 27], or on Android Intents [24, 45]. Furthermore, the majority of these analyses are concerned about Android application security rather than Android framework security.

Iannillo et al. [27] designed a gray-box fuzzer for system services. The fuzzer, *Chizpurfle*, discovers vendor-specific system service methods exposed through Binder IPC and runs a fuzzing campaign on the identified methods. To do this, it queries the system for a list of services and their methods. Based on the methods' signatures, it provides initial inputs for testing them, and mutates these inputs to produce additional test cases. Meanwhile, the system services are instrumented to monitor the test coverage. Their approach uncovered multiple bugs on a real-world Samsung phone. However, since it heavily relies on analyzing Java reflection by design, it is not applicable native system services and daemons.

Liu et al. [53] overcome this limitation, and propose a fuzzing solution, FANS, which finds vulnerabilities in native system services. However, they rely on white-box methods to discover interfaces exposed on Binder by analyzing AOSP source code. As a result, it achieves high accuracy and correctness in terms of interface identification. However, this approach renders it inapplicable to proprietary vendor-specific services where the source code is not available. Zhang et. al. [52] proposed `Invetter`, an automated static analysis tool that finds vulnerabilities in Android system services written in Java. It relies on machine learning to identify security-sensitive input validations, and uses static analysis to detect their problematic uses. Nevertheless, it suffers from the the same limitation as *Chizpurfle*, which is that it is not able to analyse native system services.

On one hand, these works clearly demonstrate their effectiveness in finding vulnerabilities in system services. On the other hand, these vulnerabilities are not coupled with a clearly defined threat model where the malicious actor is positioned with respect to the multi-layered Android security model. Thus, some of the reported vulnerabilities from these works may require chaining with other privilege escalation exploits to interact with privileged, but vulnerable Binder interfaces.

2.3.2 Unix Domain Sockets

Shao et al. [46] conducted the first study of Unix domain socket usage by both Android apps and system daemons. To perform their analysis, they developed SInspector, which identifies Unix domain socket addresses and detects authentication checks. SInspector exclusively utilizes static techniques for apps, allowing for a large scale analysis of apps. However, for system daemons, the tool needs to be run on a live, rooted Android system. They cite multiple reasons for this limitation, including the difficulty of unpacking Android factory images from different vendors, and the complexity of security enforcement for sockets, which involves the interplay of SEAndroid and DAC file permissions.

However, with the advent of new open-source tools such as the Android image unpacking library [49], and BIGMAC [25], it is now feasible to tackle these challenges and develop a fully static large-scale cross-vendor analysis framework for Unix domain socket usage in system daemons. We use these tools to overcome the challenges faced by SInspector that caused it to resort to dynamic analysis for system daemons.

2.3.3 Access Control Policy Analysis

A separate, growing body of work examines Android OS security from an access control perspective [30, 25, 50, 51, 40]. Although at the first glance this area of research might seem irrelevant, Android access control, especially SEAndroid, plays an essential role in the bigger picture of securing IPC. All Android IPC mechanisms are protected by the SE-Android policy and some are protected by DAC. Lee et al. [30] proposed a tool, PolyScope, to vet Android filesystem access control policies. They define three possible patterns of integrity violations in access control policies and rely on AOSP documentation as well as the integrity wall method to categorize process into different integrity levels. However, their work also relies on having a rooted Android phone for dynamic analysis. Additionally, their approach cannot determine which DAC UID corresponds to a process's MAC label if the process is not running. Possemato et. al. [40] conduct a longitudinal study of a large dataset of Android ROMs up to Android 9.0. Their analysis focuses on Android vendor customizations across multiple layers including SELinux policies and Android init scripts. They evaluate these customizations against Google's Compatibility Definition Document (CDD) and find a worrying number of non-compliant Android ROMs, and suggest that vendors often bypass or remove security measures added by Google in AOSP Android. Yu et al. [51] developed SEPAL, a tool that analyses SELinux policies in vendor-customized versions of Android. SEPAL uses machine learning to detect unregulated SELinux rules that can cause security vulnerabilities. However, their analysis only considers the MAC

layer. Thus, many of their alerts are unexploitable without totally compromising the DAC layer. Hernandez et al. [25] proposed BIGMAC that combines DAC and MAC to construct an attack graph representing allowed data-flows between subjects and objects in a running system. Although BIGMAC serves to abstract away the complexity of the Android security model, which was recently detailed by [33], it cannot detect DAC checks that occur dynamically in a running process.

Chapter 3

Methodology

This chapter is divided into two sections. In the first section, we describe how SAUSAGE's analysis pipeline is designed, starting from Android firmware extraction, all the way to the extraction of accessible sockets. In the second section, we discuss the specific implementation details of each step of the analysis pipeline.

3.1 Design

An overview of the architecture of the tool can be seen in Figure 1, we describe these steps in further detail below. The tool begins with the firmware image extraction. The filesystem is unpacked and extracted to acquire the SELinux policy, daemon binaries, and the init RC scripts. The SELinux policy is analyzed using a modified version of BIGMAC [25] to query the processes an untrusted app can communicate with via sockets. The results from the analysis are then compiled into a list that is used to filter the daemon binaries and the relevant service definitions we have been able to extract. The tool conducts binary analysis to verify the socket address and find any security checks that may be in the binaries. Running in parallel is the Init RC Service Definition Analysis which will extract RESERVED

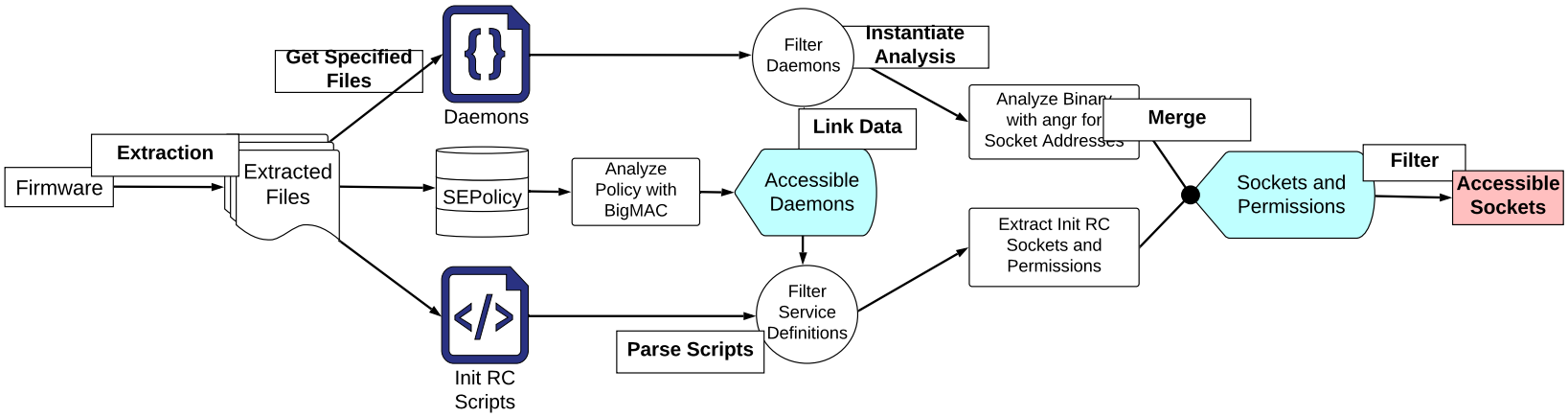

Figure 2: Framework Architecture. Following the unpacking of an image, the SELinux policy is analyzed to find the processes an untrusted app can communicate with via Unix domain sockets. The processes' binaries and their RC service definitions are retrieved from the extracted filesystem and analyzed independently to find the socket addresses that the process uses, their access control configurations, and any authentication checks in the binary code. The list of socket addresses is filtered down to the ones an untrusted app can access based on their access control permissions.

socket addresses and their file permissions from the Init RC files. The output of the binary analysis is combined with the output of the Init RC Service Definition Analysis and compiled into a list that can be used to filter for accessible sockets.

3.1.1 Image Extraction

The initial step of our tool is the firmware image extraction from a repository of firmware images that we have collected. A majority of the images have either been downloaded directly from the AOSP firmware images website [23], or from firmwarepanda [21]. The image packing format varies by vendor, e.g., HTC required an RUU Decrypt tool to account for the `.exe` format, while Samsung required `LZ4` decompression. Thus, we have to rely on existing tools that support each Android vendor and version to avoid developing our own. We use a modified version of the ATextract tool[1] to extract the files needed to analyze the SELinux policies and native daemons from the firmware images. For Android versions higher than 8.0, and for unsupported vendors, we use a modified version of a newly released unpacking tool developed for [40].

Once the images are fully extracted, the tool walks the file system to extract the DAC file permissions along with SELinux MAC labels and other Linux capabilities. The filesystem metadata is also saved since this information is used when BIGMAC generates the graphs through Init Boot Simulation and Backing File Recovery. The filesystem metadata includes Android object SELinux associations such as services, properties, and apps. It also includes the Android property list, the raw SEPolicy binary file, and the init system files which contain useful DAC/MAC/CAP metadata along with a list of native daemons started at boot.

[1] https://github.com/FICS/atcmd/tree/master/extract

3.1.2 SELinux Policy Analysis

Following extraction, our framework reasons about the system's access control policies in order to determine which processes an untrusted app can connect to through Unix domain sockets. The Android security model is based on the complex interaction of multiple security layers, including SEAndroid policies, Linux filesystem permissions and Linux capabilities. Thus, we use a modified version of BIGMAC to recreate the security state of the running system. For a full discussion of BIGMAC, we refer readers to the original paper by Hernandez et al. [25]. Once the modified version of BIGMAC finishes analyzing an image and generating the attack graph, it provides a query engine that can be used to find all the objects an untrusted app can write to. We filter the resultant list of objects to only include IPC objects of the "socket" type. Each entry in the list corresponds to a process that an untrusted app can communicate with using Unix domain sockets. Since each IPC object holds a reference to its owner process, we extract the file path of the process's binary executable, effectively obtaining the list of daemon binaries we need to analyse. The next steps of the analysis are then performed on these binaries.

3.1.3 Socket Address Extraction

Once we have the set of binary files for daemons that an untrusted app can communicate with through Unix domain sockets, we can start to extract the socket addresses that these processes are listening over. We employ two methods for each one of these daemon binaries: init RC parsing and static binary analysis. Parsing init RC files guarantees all RESERVED socket addresses will be recovered, and static binary analysis will recover all three types of socket addresses that the binary might be listening over.

```
service netd /system/bin/netd
    ...
    socket dnsproxyd stream 0660 root inet
    socket mdns stream 0660 root system
    socket fwmarkd stream 0660 root inet
```

Figure 3: Service definition of *netd* showing the `socket` option in use

Init RC Service Definitions

For each one of these binaries, there exists one or more service definitions in the Android system's init RC files. These service definitions can have options which configure how and when `init` runs these files. One of these options, `socket`,[2] creates a socket file for the service in the RESERVED namespace, creates a file descriptor for this socket and binds it to the created socket file, and saves the socket's file descriptor as an environment variable[3] for later retrieval by the service process. Figure 3 shows an example of the service definition for netd. In that case, three sockets are created with addresses dnsproxyd, mdns and fwmarkd with the specified DAC permissions. Therefore, it is straightforward to retrieve all RESERVED socket addresses from service definitions, by finding and parsing the `socket` options. However, this method does not capture socket addresses in other namespaces.

Static Binary Analysis

In the binary analysis module, we first construct the Control Flow Graph (CFG) of the binary and all externally linked objects, and identify all defined functions. We then perform an inter-procedural dataflow analysis starting at the entry point of every function that calls

[2]The `socket` option follows the syntax: `socket <name> <type> <perm> [ <user> [ <group> [ <seclabel> ] ] ]` where `<name>` is the address of the socket, `<perm>`, `<user>` and `<group>` are its credentials on the filesystem, and `<seclabel>` is its SELinux label

[3]This environment variable's name is formatted as `ANDROID_SOCKET_<address>` where `<address>` is the address of the socket in the reserved namespace

the `bind` system call in the binary. At the `bind` callsite, we extract the value of the address argument. The address is checked to determine whether or not it is a Unix domain socket address. If it is, we detect the namespace that the address belongs to by checking the first character of that address. If it is a null byte, then it belongs to the ABSTRACT namespace and no further analysis is needed. If the address starts with a directory separator ('`/`'), then it belongs to the FILESYSTEM namespace, and we attempt to determine the permissions the socket file is created with. This is done by detecting all preceding `umask`, `seteuid` and `setegid` system calls in the binary and extracting their arguments. The same process is carried out for subsequent `chmod`, `fchmod`, `chown`, and `fchown` calls. Additionally, the static binary analysis module detects RESERVED socket addresses by performing the same type of dataflow analysis for functions that call `getenv`. If the requested environment variable name starts with the "`ANDROID_SOCKET_`" prefix, the rest of the environment variable name is saved as a RESERVED socket address.

3.1.4 Peer Credential Check Extraction

The `getsockopt` system call [32], when invoked with a file descriptor `sockfd`, retrieves the value of various options for the socket pointed to by `sockfd`. The option retrieved is specified by the `optname` argument, which is an integer corresponding to a valid socket option. The retrieved option is stored in the pointer specified by the `optval` argument. In our case, we are mainly interested in `getsockopt` calls where the `optname` is specified as `SO_PEERCRED` (0x11). In this case, the connected peer's credentials are stored at the `optval` pointer in a `ucred` struct. This struct contains three member variables: the Process ID (PID), UID and GID of the connected peer.

Using the same CFG used in the Socket Address Extraction step, we perform another dataflow analysis of every function that invokes the `getsockopt` system call. First, we check the value of the option name (`optname`) argument. If the function is called with

the `SO_PEERCRED` optname, we track all subsequent uses of the returned credentials in the function and record which credentials are being used. We use the same categorization used in [46]; UID- and GID-based checks are considered secure, while PID checks are considered weak. Additionally, we attempt to detect and categorize uses of these credentials. We have identified two types of uses: (1) *Integer comparisons:* We detect whenever a credential is used in a comparison instruction and record the operand if it is a constant integer, or "UNDEFINED" if it is not. (2) *Function arguments:* We detect whenever a credential is used as a function argument and record the function address and name (if it was not stripped). With this usage information, we can determine exactly what credentials a connected peer needs to be able to communicate with the process through a given socket, thus greatly aiding in further interpretation of the analysis result.

3.1.5 File Permission Analysis

Following the detection of all socket addresses and their filesystem permissions (if any), we check whether an untrusted app with all possible permission-mapped GIDs can access the socket file for each FILESYSTEM or RESERVED socket address. We use a modified version of BIGMAC to reason about the MAC policy and determine whether an untrusted app has access to a socket file. We also inspect the socket file's permission bits, UID and GID to determine access w.r.t. the DAC policy. This analysis is not applicable to ABSTRACT socket addresses in our result set as they are fileless and are therefore accessible regardless.

3.2 Implementation

In this section, we discuss the implementation of the three most crucial parts of the framework architecture, which includes the extension of BIGMAC to better serve our use case, the static analysis of daemon binaries using `angr` [47], and the final step of filtering and

categorizing accessible sockets.

3.2.1 Initial BigMAC Query

The first step following successful extraction of a firmware image is the SELinux policy analysis. We implemented an easy-to-use API that exposes the most crucial functionalities of BIGMAC to the developer. This API facilitates running the whole workflow of BIGMAC in a single call, and implements an interface to the prologue engine that facilitates query operations on the generated Attack Graph. We use this API by specifying the path of the extracted SELinux policy and running the attack graph instantiation. Using this attack graph, we run the query: `query(untrusted_app, _, 1)` to retrieve all nodes in the graph that an untrusted app can write/connect to. We then filter only socket objects from the resultant list. Since socket objects are `IPCNodes`, they hold a reference to the owning process. Thus, we can retrieve all the processes, and their executable binaries, that an untrusted app can connect to through a Unix domain socket.

Additionally, we extended BIGMAC's Init Boot Simulation step to handle `socket` options under service definitions in init RC files. On encountering a `socket` entry under a service definition, BIGMAC now creates the corresponding file in the simulated filesystem as part of the boot process with the specified permissions. Additionally, it assigns the correct SELinux context to the socket file based on the extracted filesystem contexts. This addition is essential as BIGMAC removes filesystem contexts that do not have a backing file from the attack graph which would have prevented us from querying whether an untrusted app has access to these socket files.

3.2.2 Static Binary Analysis

Our static binary analysis implementation was implemented in ~2K LoC and contains three modules. The Socket Address Extraction module extracts socket addresses that the binary

Function	Namespace	Library
FrameworkListener	RESERVED	libsysutils.so
SocketListener	RESERVED	libsysutils.so
android_get_control_socket	RESERVED	libcutils.so
socket_local_server_bind	Any	libcutils.so
socket_local_server	Any	libcutils.so

Table 2: Android-specific bind APIs

is listening over by analyzing `bind` call sites. The DAC Check Extraction module detects and analyzes DAC checks by performing data flow analysis after `getsockopt` calls with the `SO_PEERCRED` argument as discussed in Section 3.1.4. Each daemon binary an untrusted app can connect to is statically analyzed to retrieve all the Unix domain socket addresses it listens on and the permissions they are created with, as well as detect any hardcoded DAC checks in the binary.

We implement our static binary analysis using the `angr` framework [47]. We first generate the CFG of the analyzed binary during which function prologues are detected and stored in the `angr` project's knowledge base. The dataflow analysis is implemented by `angr`'s intra-procedural `ReachingDefinitions` analysis [15] and is used in both socket address extraction and DAC check extraction. To make the analysis inter-procedural, we implement a `FunctionHandler` which handles function calls by performing the `ReachingDefinitions` analysis recursively based on [39].

Socket Address Extraction

First, we find all call sites to the `bind` system call. This is done by finding the `bind` function node in the CFG and listing all of its predecessor nodes where the connecting edge is of type `Ijk_Call`, signifying a function call. For each one of these nodes, we find the function that it belongs to, and perform an inter-procedural data flow analysis on that function. The `FunctionHandler` also simulates common libc string manipulation functions, such

as `strcpy`, `sprintf` and others, in order to capture dynamically constructed socket addresses at the `bind` callsite. These string manipulation handlers are implemented in order to avoid inaccuracies caused by the complex control flow structures associated with string operations. Additionally, Android provides additional utility APIs for system daemons to create, bind, and listen over local sockets. These functions are found in `libcutils.so` and `libsysutils.so` and are detailed in Table 2. We perform the same analysis at the call sites of these functions to recover socket addresses passed to these utility functions.

DAC Check Extraction

The same dataflow analysis implementation is used for DAC check extraction. We analyze call sites of the `getsockopt` system call, and check its arguments. If the `optname` argument is set to `SO_PEERCRED`, we track usages of the `ucred` struct, stored in the pointer `optval`, by tainting its member variables. We record any variables used and attempt to identify the type of usage as discussed in Section 3.1.4.

3.2.3 File Permission Analysis

We add additional functionality to BIGMAC allowing us to add previously undetected files, e.g., files created dynamically by a running process. Following the recovery of FILESYSTEM socket addresses and their DAC metadata, we insert these filenames along with their metadata into BIGMAC's recovered filesystem, and rerun BIGMAC's workflow. This will assign the correct SELinux labels to these files in an automated manner, which allows us to determine whether a socket file is accessible through simple queries of the Prolog engine.

Chapter 4

Results

In this chapter, we first present our findings on accessible sockets, and security downgrade; we then report on the performance measurements and ground truth comparison. The number of firmware images analyzed by vendor and version can be found in Figure 4. Accessible sockets are the sockets that we have identified to be accessible for an untrusted app without any prerequisites (following our threat model). For security downgrade, we consider daemons that exist in AOSP Android but have weaker security protections due to vendor customization of access control policies and customization of the daemons themselves. Additionally, we discuss the insecure usage of ABSTRACT sockets we encountered in our dataset. Lastly, we perform case studies of interesting findings uncovered by our analysis that lead to the discovery of vulnerabilities. In Figure 4, we show the distribution of Android versions and vendors in our dataset, and we present Figure 5 which displays the different amounts of accessible system daemons by vendor.

4.1 Accessible Sockets

We identified 28 unique socket addresses that an untrusted app is allowed to connect to by the access control policy in at least one of the firmware images analyzed. We present

Address	Namespace	System Daemon	Authentication Checks	Vendor
logd	RESERVED	logd	None	All
logdr	RESERVED	logd	None	All
logdw	RESERVED	logd	None	All
dnsproxyd	RESERVED	netd	None	All
fwmarkd	RESERVED	netd	None	All
pdx/system/vr/display/client	RESERVED	surfaceflinger	None	All
pdx/system/vr/display/manager	RESERVED	surfaceflinger	None	All
pdx/system/vr/display/vsync	RESERVED	surfaceflinger	None	All
tombstoned_java_trace	RESERVED	tombstoned	None	All
traced_producer	RESERVED	traced	None	All
perfd	RESERVED	perfd	None	All
/dev/socket/nims	FILESYSTEM	cnd	None	Asus, HTC, Motorola, Xiaomi
cnd	RESERVED	cnd	<GID>OR <AppName>	Asus, HTC, Motorola, Xiaomi
qvrservice	RESERVED	qvrservice	None	Asus, HTC, Samsung, Xiaomi
seempdw	RESERVED	seempd	None	Asus, HTC, Xiaomi
@fmhal_sock	ABSTRACT	fmhal_service	<UID>	Asus, Motorola, Xiaomi
@qcom.dun.server	ABSTRACT	dun-server	None	Asus, Xiaomi
@cand.socket.ctrl	ABSTRACT	cand	None	HTC
@cand.socket.msg	ABSTRACT	cand	None	HTC
dmagent	RESERVED	dmagent	None	HTC
cfiat	RESERVED	rild	UID	HTC
kipc	RESERVED	rild	UID	HTC
/dev/socket/dpmwrapper	FILESYSTEM	dpmd	None	HTC, Xiaomi
@btloggersock	ABSTRACT	bt_logger	None	Motorola, Xiaomi
@dev/socket/jack/set.priority	ABSTRACT	apaservice	None	Samsung
napproxyd	RESERVED	netd	None	Samsung
tcm	RESERVED	dpmd	None	Xiaomi
dpmwrapper	RESERVED	dpmd	None	Xiaomi

Table 3: Socket addresses an untrusted app can connect to, their system daemons, authentication checks they implement and the vendors where they were found to be accessible

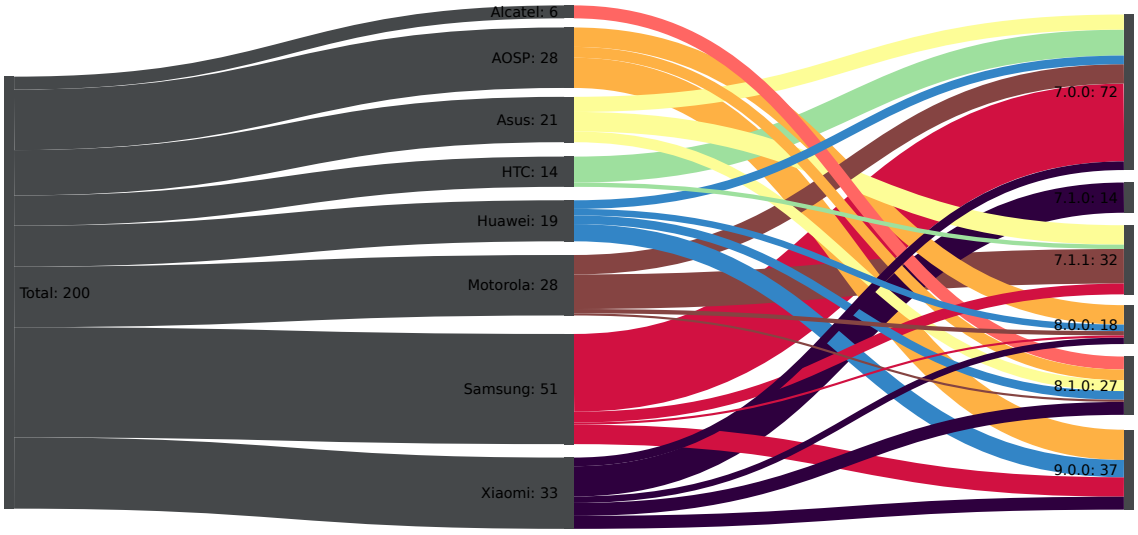

Figure 4: Distribution of Android versions and vendors in our corpus

17 of these socket addresses and their daemons in Table 3. We also specify any authentication checks performed on the client after a connection is established, as well as the list of vendors these socket addresses were detected on.[1] In the following, we discuss each daemon's functionality and its accessible sockets. Information about proprietary daemons' functionality is not publicly available. Therefore, we infer their functionality based on static analysis of the binaries, as well as whatever information we can find online.

4.1.1 AOSP Daemons

AOSP daemons are available in all Android systems as part of AOSP Android. The AOSP version of these daemons' source code is made available by AOSP, although vendors might make proprietary customizations to them in their own distributions of Android. These daemons purposefully expose sockets for communication with an untrusted app to implement various functionalities. These sockets were detected consistently across our dataset, confirming our tool's reliability at detecting accessible sockets. Since these sockets are

[1]We only specify the vendors with firmware images where the system daemon that uses these sockets is *accessible*. If the daemon exists in other vendors' firmware images, we do not include it.

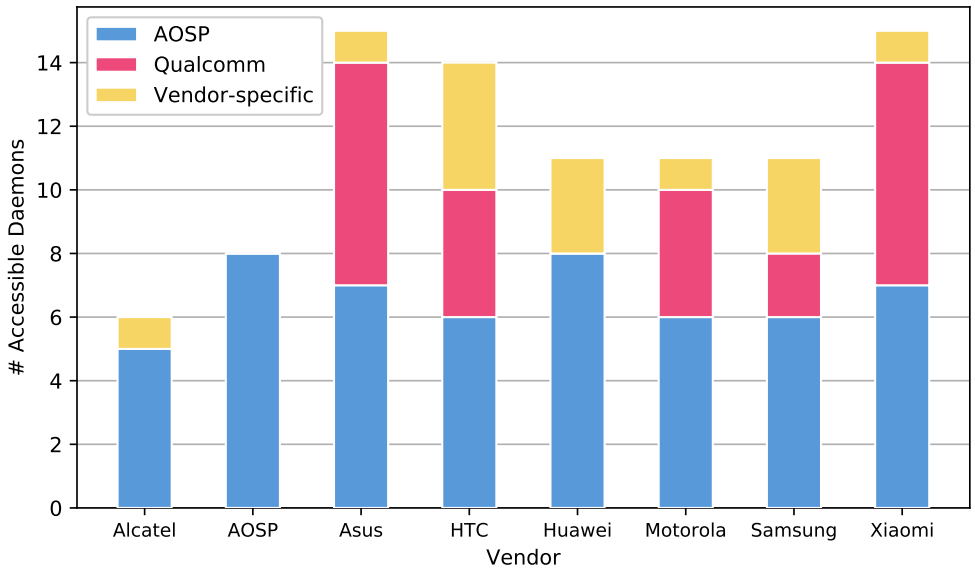

Figure 5: Accessible system daemons by vendor in our corpus

intended to be accessed by untrusted apps, we omit these accessible sockets from Table 3, and briefly discuss the daemons' functionality instead.

The *logd* daemon is a centralized logger implementing all logging operations in Android [31]. It utilizes three sockets, all of which are accessible to an untrusted app: logd, logdr, and logdw. The *netd* daemon is responsible for managing network interface configurations [**?**]. In AOSP Android, the *netd* daemon utilizes four socket addresses: netd, dnsproxyd, mdns and fwmarkd. Of these sockets, dnsproxyd and fwmarkd are accessible to an untrusted app with the `INTERNET` permission. The *surfaceflinger* daemon's main functionality is to compose and render multiple display surfaces onto the display [31]. In Android 8.0+ images, we found three accessible RESERVED socket addresses from the *surfaceflinger* daemon binary located in the pdx/system/vr/display/ directory and include client, manager, and vsync. The *tombstoned* daemon was added in Android 8.0 and it plays a role in capturing crash data from a system and storing it for further analysis. The *traced* daemon is part of an open-source solution developed by Perfetto [9] and used in Android

for system profiling, app tracing and trace analysis. The *perfd* daemon collects information to keep track of performance on the system.

4.1.2 Qualcomm Daemons

Qualcomm provides a wide range of hardware and peripherals on Android devices, such as the processor and the Mobile Station on Modem (MSM) System on Chip (SoC). For interoperability between these peripherals and the operating system, Qualcomm implements daemons that bridge the communication between these devices and the rest of the Android framework. These daemons were found to be accessible across multiple vendors' images in our dataset. In AOSP however, these daemons exist, but none of them are accessible to an untrusted app. This discrepancy indicates that access control policies placed by AOSP were relaxed by other vendors where these daemons were found to be accessible.

cnd. The *cnd* daemon manages Qualcomm Connectivity Engine which chooses between 3G/4G and Wi-Fi networks based on their performance for the specific application a user is using [42]. It is a proprietary daemon, therefore its exact functionality is not publicly known. In Android versions prior to 8.0, it uses the cnd RESERVED socket and a FILESYSTEM socket located in /dev/socket/nims, both of which are accessible to an untrusted app with the `INTERNET` permission. The /dev/socket/nims socket was also found to be world-accessible in some HTC, Motorola and Xiaomi images. It is unclear whether this is a result of misconfiguration or a change of the socket's functionality.

qvrservice. The *qvrservice* daemon is a proprietary daemon that manages Qualcomm VR service. It exposes a world-accessible RESERVED socket qvrservice.

seempd. The *seempd* daemon is part of the Qualcomm Trusted Execution Environment (QTEE) [43]. It exposes a world-writable DGRAM socket with address seempdw.

dpmd. The *dpmd* is a daemon which stands for Data Port Mapper, and is part of the

QTI DPM Framework [4]. It is unclear what functionality it provides. This daemon utilizes two sockets: a RESERVED socket with address dpmd and a FILESYSTEM socket with address /dev/socket/dpmwrapper. The dpmd socket is configured to be inaccessible to apps. In Android 8.0 images, it uses an additional RESERVED socket named tcm, and the FILESYSTEM socket /dev/socket/dpmwrapper was changed to a RESERVED socket "dpmwrapper". The dpmwrapper and tcm RESERVED sockets require the `INTERNET` permission to be able to connect.

dun-server. *dun-server* is a daemon that implements and manages Dial-Up Networking over Bluetooth [3]. It listens over the @qcom.dun.server ABSTRACT socket address. It contains no authentication check, allowing any client to connect to dun-server over this socket.

fmhal_service. *fmhal_service* is an open-source daemon that manages FM radio on supported systems [2]. It exposes an ABSTRACT socket with address @fmhal_sock. Since this socket is ABSTRACT, it is accessible by default. Thus, any app can establish a connection to it. However, when a client connects, a UID check is performed to ensure that the client's UID is one of root, system or bluetooth.

bt_logger. *bt_logger* is an open-source daemon that has the ability to log Bluetooth traffic [18]. It exposes an ABSTRACT socket with address @btloggersock with no authentication check, allowing any client to start/stop Bluetooth snooping.

4.1.3 Vendor-specific Daemons

Vendor-specific daemons are daemons that are developed by the Android device manufacturer and bundled with their operating system distribution. A daemon is classified as vendor-specific if it is present in the Android images of a single vendor. In our results, two out of three accessible vendor-specific daemons run as root, presenting valuable targets for exploitation. The third daemon provides an interface to a function available only

to processes running privileged UIDs.

dmagent. HTC *dmagent* is a proprietary daemon that manages the Open Media Alliance Device Management protocol. dmagent runs with UID root and listens over the RESERVED socket address dmagent. The socket is configured to be accessible to applications with the `INTERNET` permission. This socket has been previously used to issue copy file command to the daemon which acts to copy files as root from arbitrary source to arbitrary destination [28]. Our analysis shows that this socket remains unprotected by access control policies or authentication checks, making it a prime target for exploitation of a root process.

cand. HTC *cand* is a proprietary daemon which runs as root and listens over two ABSTRACT socket addresses: @cand.socket.ctrl and @cand.socket.msg. Through static analysis, we determined that the daemon serves as an interface to communicate over the CAN-Bus, although the specific use case is not clear. Nevertheless, since it runs as root and listens over unprotected ABSTRACT sockets, it may present another security risk much like dmagent.

apaservice. The *apaservice* daemon is part of the Samsung Android Professional Audio framework [34]. It runs as with a UID of "jack." We found that it creates and listens over one ABSTRACT socket address @dev/socket/jack/set.priority which is used as an interface through which it calls `android::requestPriority` with the parameters it receives. According to AOSP source code [13], this functionality should only be exposed to processes with the audioserver, cameraserver and bluetooth UIDs.

4.2 Downgraded Security

A system daemon is considered to have downgraded security if the vendor relaxes SELinux rules that would have prevented communication between the daemon and an untrusted app in AOSP. To find these instances of downgraded security, we go over the list of daemons an untrusted app can communicate with for each vendor. We then try to find each daemon

```
431  # HTC-htc_dk-BEGIN: for dumping kernel log from system_server
432  /dev/socket/htc_dk                              u:object_r:htc_dk_socket:s0
433  # HTC-htc_dk-END
434  # HTC-htc_dlk-BEGIN: for dumping last kernel log from system_app
435  /dev/socket/htc_dlk                             u:object_r:htc_dlk_socket:s0
436  # HTC-htc_dlk-END
```

Figure 6: Comments in `file_contexts` in HTC Firmware show usage of the htc_dk and htc_dlk sockets [16]

in that list and its service definition in the corresponding AOSP Android version. If the service exists in AOSP and is enabled, but is not accessible by an untrusted app, then we flag it is as a security downgrade.

HTC dumpstate. The *dumpstate* system daemon is an AOSP system daemon that can generate logs that are used to collect details of device-specific issues; an untrusted app is disallowed to communicate with this daemon in AOSP. HTC relaxed this restriction and added two extra sockets to the daemon: htc_dk and htc_dlk. Untrusted app access to these sockets is not allowed by both the MAC and DAC policies. However, as per the comments of the `file_contexts` file shown in Figure 6, the htc_dlk socket sends kernel log messages to a system app. This is a bad security practice as kernel logs can hold sensitive information, and pre-installed apps packaged with vendor-customized firmware have been shown to be insecure [19].

HTC rild. *rild* is the Radio Interface Layer daemon in Android [1, 14]. It provides an abstraction layer between Android telephony services layer and the radio hardware layer and handles all telephony operations such as call handling, SMS, and others. In Android versions prior to 8.0, *rild* utilizes three sockets: rild, rild-debug and sap_uim_socket1. In AOSP Android, communication with rild using Unix domain sockets is not allowed for untrusted apps by the SELinux policy. In HTC images, our SELinux policy analysis showed that the policy was relaxed and an untrusted app was allowed to communicate with *rild*. Furthermore, we detected two vendor-specific sockets, kipc and cfiat, both of which

grant read and write access to an untrusted app with the `INTERNET` permission. These socket addresses have only been detected on the HTC firmware images we analyzed, and their file contexts are labeled htc_cfiat_socket and htc_kipc_socket, confirming that they originate from an HTC-specific vendor customization of *rild*. We detected a UID-based authentication check in the HTC *rild* binary, leaving these sockets potentially protected only by a single post-connection DAC check. Therefore, if a malicious app changes its UID through a privilege escalation exploit, it can gain access to these sockets, which would not have been possible if the SELinux policy had not been relaxed.

cnd. The *cnd* daemon can be available in AOSP firmware images. In all of the AOSP and Samsung images tested, an untrusted app does not have access to this daemon. On the other hand, Asus, HTC, Motorola and Xiaomi firmware images allow an untrusted app to communicate with this daemon. In this case, the daemon exposes two sockets: cnd and /dev/socket/nims that are accessible to an untrusted app, one of which has no authentication checks. Both require the app to have the `INTERNET` permission.

Asus mm-qcamera-daemon. In Asus Android images, *mm-qcamera-daemon* is allowed to communicate with an untrusted app. It contains a socket named at address /data/misc/camera/cam_socket in the FILESYSTEM namespace. The socket itself is inaccessible by both the MAC and DAC policies.

4.3 Abstract Socket Denial of Service

Our analysis aims to find sockets accessible to untrusted apps. However, we report on the ABSTRACT sockets in our results we found are vulnerable to DoS. ABSTRACT socket addresses are vulnerable to DoS if the daemon closes the socket at any point in its operation, or if the daemon exits for any reason. We detect the first case through manual static analysis using Ghidra [35]. This is done by detecting `close` calls on the socket that was previously bound to an ABSTRACT address. If `close` is called anywhere outside of a final cleanup

of the daemons resources during termination, then we flag it as vulnerable to DoS.

As for the second case, we detect daemons which are started or stopped by property triggers. Property triggers are defined in Init RC files, and are used to start/stop a service daemon when the specified system property changes, depending on the value of the system property. As a result, if a property trigger causes the system daemon to be stopped, a malicious app can occupy the ABSTRACT address. When the daemon is restarted by another property trigger, it fails to bind the socket to its ABSTRACT address.

Four of the ABSTRACT sockets we found are vulnerable to DoS as they match these criteria: @dev/socket/jack/set.priority, @fmhal_service, @qcom.dun.server and @btloggersock. @dev/socket/jack/set.priority is bound by *apaservice* and is triggered by a service call to `IAPAService::StartJackd`. Therefore, a malicious app can occupy this ABSTRACT socket address before another app makes the service call. When the service call is made, *apaservice* would fail to bind the socket. However, this failure is handled gracefully by *apaservice* so that its other functionalities would remain unaffected. @fmhal_service and @btloggersock are bound by their system daemons on initialization, but the daemons themselves are triggered by a property trigger in init RC files. A malicious app can bind either of these addresses while the corresponding property is not set. When the property is set and the daemons are started, they fail to bind the socket address and terminate due to the resulting bind error. @qcom.dun.server is bound and closed repeatedly by *dun-server* after every connection. Exploiting this DoS vulnerability would require a race-condition, where a malicious app attempts to bind the @qcom.dun.server address before *dun-server* re-binds it.

4.4 Case Studies

In this section, we focus on two interesting cases from our results that translate to vulnerabilities that a malicious app can potentially exploit. We discuss Samsung *apaservice* daemon's FILESYSTEM socket and how it can be used to request a priority for any process ID or thread ID, a functionality only available to processes with the audioserver, cameraserver or bluetooth UID. We discuss another potential permission bypass exploit in Qualcomm's *cnd* and *dpmd* daemons, both of which implement an identical check based on the connecting process's name.

4.4.1 Samsung apaservice

Our analysis indicates that Samsung's *apaservice* daemon creates an ABSTRACT socket with address @dev/socket/jack/set.priority. In our ground truth evaluation, this socket was not found at first on the running device. We analyzed the *apaservice* binary to determine the reason, and found that this socket is only created after calling `APAService::startJackd`. This method is exposed by the service `com.samsung.android.jam.IAPAService`. After calling this method using the `service call` command, the socket was created and appeared in the `netstat` output. This demonstrates the effectiveness of our analysis in uncovering sockets which are created only under certain conditions, as compared to dynamic analysis. Aside from the DoS discussed in Section 4.3, this socket exposes two other, more critical vulnerabilities.

Authorization Bypass

The socket at @dev/socket/jack/set.priority is a DGRAM socket under *apaservice* that accepts messages from any client, with no DAC check after receiving a message. It receives messages of the format "`*4<pid>,<tid>,<priority>`," where `<pid>` is a process ID, `tid` is a thread ID, and `<priority>` is the requested priority. These values are then

passed to the function `android::requestPriority`, which requests the SchedulingPolicyService to assign a priority to the requested process ID and thread ID. Although this request is typically available to system processes running under the audioserver, cameraserver and bluetooth UIDs, an untrusted app can set its own priority, or the priority of any other process through this socket, effectively bypassing authorization checks.

Local Privilege Escalation

Additionally, through manual analysis, we discovered that the function that handles messages received over this socket, `android::APAService::handlePriorityMessage`, is vulnerable to buffer overflow. By sending the correct preamble, "4*", followed by 25 bytes of data, the *apaservice* daemon crashes due to stack corruption. The backtrace logs show that the return address was successfully overwritten. The impact of this buffer overflow vulnerability can range from DoS of *apaservice* to local code execution as the "jack" user in a less restrictive SELinux context. We developed a Proof of Concept (PoC) and sent it to Samsung as part of responsible disclosure. Our PoC crashes the daemon, achieving DoS. Achieving local code execution would lead to the malicious application achieving local privilege escalation to the `apaservice` SELinux context and the jack UID, which *apaservice* runs with. However, this would require bypassing the buffer overflow protections compiled into the daemon binary. Samsung confirmed this vulnerability and rewarded our findings through its bug bounty program.

4.4.2 Qualcomm cnd and dpmd

In 18 Xiaomi and four HTC images analyzed, we found that *dpmd* implements an insecure authentication check after a connection is established. The *cnd* daemon implements an identical authentication check in 14 HTC images and 7 Motorola images. Both of these

daemons are started by init with root UID, but later drop to system UID through a `setuid` call. The authentication mechanism works as follows: after a connection with a new client is established, the client's DAC credentials are retrieved using a `getsockopt` call. First, the GID is compared against a list of GIDs that are allowed by the daemon. If the GID does not match any of the allowed GIDs, then the PID is used to retrieve the connecting client's process name by reading the `/proc/<pid>/comm` file for that process. In Unix systems, this file exists for every process and contains the process name. The process name is then compared to a list of allowed process names, and access is granted if a match is found. This is however an insecure check, as any app can change its own process name dynamically, even if a different process has the same name. Therefore, a malicious app can bypass this check trivially by changing its name to that of an allowed process.

In the case of *dpmd*, this check is implemented for an inaccessible RESERVED socket of the same name, "dpmd," and an untrusted app is not allowed to connect to by both MAC and DAC. On the other hand, *cnd* implements this check on its "cnd" RESERVED socket, which is accessible to an untrusted app. Through static analysis, we infer that this socket allows clients to get/set network settings such as WiFiAP, WiFi P2P, and Default Network settings, by sending the appropriate command over the cnd socket.

Chapter 5

Tool Evaluation and Limitations

In this chapter, we establish the correctness of SAUSAGE by comparing it to the ground truth from three different physical Android devices. Moreover, we evaluate the tool's performance in terms of time taken per image or each vendor. Finally, we identify the limitations and challenges we face in developing our tool and in applying it to a diverse dataset.

5.1 Ground Truth Evaluation

To confirm the correctness of our framework in detecting sockets and their access control properties, we ran a ground truth evaluation on Samsung devices running Android 7.0 and 8.0, and a Motorola device running Android 7.1.1. The test was carried out by an app we developed which obtains the permissions we listed in our threat model. The app runs the `netstat -xl` command to list all the listening Unix domain sockets and their addresses. The app then tries to connect to each socket address and displays a table containing the socket addresses and the result of the connection attempt. However, this app is not part of our framework and only serves to collect ground truth data for evaluation.

On all devices, the app successfully connected to the fwmarkd, dnsproxyd, logd, logdr RESERVED socket addresses. On the Motorola device, the app also reported

a connection to the perfd RESERVED socket address. All of these socket addresses were accurately detected by our framework as accessible sockets. For the Samsung device, our analysis detected an ABSTRACT socket owned by the *apaservice* daemon at @dev/socket/jack/set.priority which was not found on the running device. We later discover that the socket is created only after a Samsung-specific service is activated through a Binder call and discuss the details in Section 4.4.1. This demonstrates the effectiveness of our approach compared to dynamic analysis which may not detect sockets created conditionally or in response to a trigger. Note that our ground truth evaluation app is a simplified implementation of the *Connection Tester* dynamic analysis module used in [46]. Thus, we claim that our approach achieves a better socket detection rate due to the higher coverage inherent to static analysis.

5.2 Performance Evaluation

We ran our analysis on a PC with the Intel(R) Core(TM) i7-8700 CPU @3.20GHz and 16 GB RAM. On average, each firmware was analyzed in 770 seconds (~13 minutes). Figure 7 shows a box plot of the time taken for each firmware image. The static binary analysis takes a majority of that time at an average of 736 seconds (~12 minutes), resulting in a large variance for each image, depending on how many binaries are being analyzed, i.e., how many service daemons are accessible to an untrusted app, and how complex their binaries are. The remaining time is used in the initial and final steps in the instantiation and querying of BIGMAC. Currently, our prototype does not implement obvious optimizations, such as parallelization of the binary analysis step for each binary. We leave these engineering tasks as future improvements to our framework. Based on monitoring the memory usage for our analysis, parallelization would have resulted in a 3x speedup on the same PC.

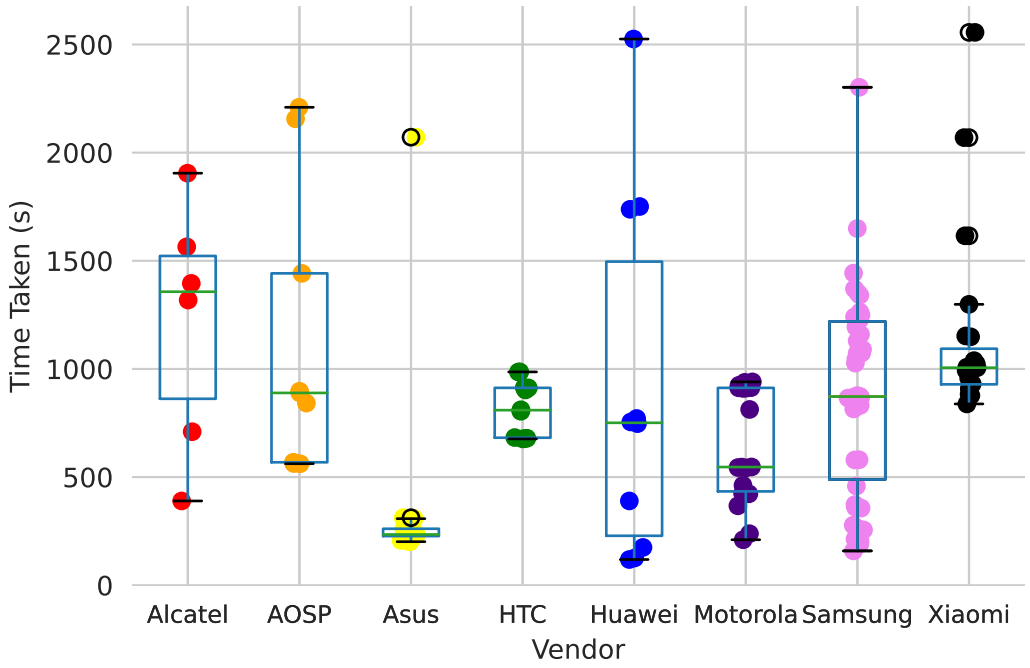

Figure 7: Analysis time for firmware images in our corpus

5.3 Limitations

Our analysis approach faces certain limitations. Firmware unpacking and extraction presents the only obstacle to expanding our analysis to more recent Android versions and a wider variety of vendors. Extending the current open-source toolset for Android image extraction requires significant engineering effort, but can pave the way for similar large-scale analyses. Additionally, we discuss the limitations inherent in our static binary analysis approach that make the analysis of statically-linked stripped binaries difficult.

5.3.1 Firmware Unpacking and Extraction

Extracting and unpacking Android images is not trivial as the format of factory images can vary greatly between different Android vendors and versions. Multiple tools have been developed that facilitate the unpacking process or different stages of it. However, to

our knowledge, there is no freely available unified factory image unpacking tool that can unpack any firmware image across different versions and vendors, except for the ones we used [49, 40]. These tools are outdated however, and only support Android versions 5-9 for AOSP, and 5-8 for other vendors. Furthermore, within these versions the unpacking success rate is not perfect, and some filesystems may not be recovered. This limits the operable dataset we can use in our analysis, and as the result extracted firmware might have missing daemons or SELinux policy files.

5.3.2 Static Binary Analysis

Our implementation of the static binary analysis relies on detecting Android bind APIs and string manipulation functions by their symbol name. This does not pose a problem in the case of dynamically-linked binaries since external symbols are persevered for linking. However, this becomes problematic in statically-linked stripped binaries. In our analysis, we encountered three cases of statically-linked stripped binaries which we ultimately skipped, namely: *mcDriverDaemon*, *debuggerd* and *adbd*. Additionally, we assume that if a `bind` call exists in the binary with a Unix domain socket address parameter, then that socket is bound at some point by the daemon of the daemon. We do not perform reachability analysis to avoid the problem of inaccurately resolving indirect jumps.

Chapter 6

Concluding Remarks and Future Work

In this work, we presented SAUSAGE, a static analysis framework to evaluate the security of Unix domain sockets used in Android. Our approach combines fine-grained access control policy analysis with static binary analysis techniques to comprehensively detect exposed IPC sockets available to an untrusted app. We use this framework to analyze 200 Android images from different vendors and Android versions, and uncover vulnerabilities and access control misconfigurations, such as permission bypass and denial of service. Some of these sockets would not have been discovered by previous work relying on dynamic analysis.

For future work, we plan to use our framework to discover vulnerabilities stemming from other types of misuse than misconfigured access control. For instance, the results described in Section 4.3 are more prevalent than we expected even though they were not the main focus of our analysis. Previous research [46] assumed that system daemons create ABSTRACT sockets on initialization, and never close them as long as the system is running. Therefore, this issue was overlooked as a malicious app would not be able to occupy the same ABSTRACT sockets. However, our findings show that some daemons do in fact close these sockets or stop entirely in response to a property change, allowing a malicious app to occupy the daemon's ABSTRACT socket addresses. The impact of this misuse of

ABSTRACT sockets can range from DoS to a malicious application spoofing a system daemon in confused deputy-like attacks. This provides a strong indication that repurposing our framework to discover ABSTRACT sockets in general, and not only the ones accessible by an untrusted app, will be fruitful.

Additionally, we plan to identify systematic methods of distinguishing between misconfigured unprotected sockets and intended accessible sockets. This would allow for automatic labelling of vulnerable unprotected sockets and reduce the manual analysis effort. To strengthen our vulnerability discovery component, we are considering adding a taint analysis module, similar to the one used by Redini et al. [44], to discover memory corruption vulnerabilities exposed by sockets, such as the one described in Section 4.4.1. Alternatively, we can resort to fuzzing of exposed sockets after using SAUSAGE to discover their addresses, adopting the same approach used by efforts studying the Binder IPC [27, 53].

Bibliography

[1] Radio Layer Interface | Android Open Source Project. https://wladimir-tm4pda.github.io/porting/telephony.html, Sep 2015.

[2] fmhalservice · Codeaurora / platform_vendor_qcom-opensource_fm. https://gitlab.com/Codeaurora/platform_vendor_qcom-opensource_fm/-/tree/44d045c660bcfd9063a77cdfce3694c8f5b9dbef/fmhalService, 2017.

[3] src/org/codeaurora/bluetooth/dun · Codeaurora / platform_vendor_qcom-opensource_bluetooth. https://gitlab.com/Codeaurora/platform_vendor_qcom-opensource_bluetooth/-/tree/db31ce2e09daeb551320b4f9d20e56000123edd0/src/org/codeaurora/bluetooth/dun, 2017.

[4] Kang Netmgr QMI DPM (Data Port Mapper) from Xiaomi Daisy. https://review.lineageos.org/c/LineageOS/android_device_lenovo_YTX703-common/+/239742, Feb 2019.

[5] Manifest.permission | Android Developers. https://developer.android.com/reference/android/Manifest.permission, Jun 2021.

[6] Mobile & Tablet Android Version Market Share Worldwide. https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide, 2021.

[7] Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, 2021.

[8] Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share, 2021.

[9] perfetto | Android Developers. https://developer.android.com/studio/command-line/perfetto, May 2021.

[10] S. Ali, M. Elgharabawy, Q. Duchaussoy, M. Mannan, and A. Youssef. Betrayed by the Guardian: Security and Privacy Risks of Parental Control Solutions. In *Annual Computer Security Applications Conference*, 2020.

[11] S. Ali, M. Elgharabawy, Q. Duchaussoy, M. Mannan, and A. Youssef. Parental Controls: Safer Internet Solutions or New Pitfalls? *IEEE Security and Privacy*, (1):2–12, May 2020.

[12] S. Ali, M. Elgharabawy, Q. Duchaussoy, M. Mannan, and A. Youssef. Betrayed by the Guardian: Security and Privacy Risks of Parental Control Solutions. In *IEEE European Symposium on Security and Privacy - Invited Poster*, 2021.

[13] Android Open Source Project. SchedulingPolicyService.java. `https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/os/SchedulingPolicyService.java`.

[14] Android Open Source Project. RIL Refactoring. `https://source.android.com/devices/tech/connect/ril`, Sep 2020.

[15] Angr.io. Angr 9.0.7912 documentation. `https://angr.io/api-doc/angr.html#angr.analyses.reaching_definitions.reaching_definitions.ReachingDefinitionsAnalysis`.

[16] Bigcountry907. Kernel_htc_a32eul/file_contexts at master · bigcountry907/kernel_htc_a32eul. `https://github.com/Bigcountry907/kernel_htc_a32eul/blob/master/toolz/boot/components/ramdisk/file_contexts`, Oct 2016.

[17] C. Cao, N. Gao, P. Liu, and J. Xiang. Towards Analyzing the Input Validation Vulnerabilities Associated with Android System Services. In *Annual Computer Security Applications Conference*, 2015.

[18] Codeaurora. bt_logger/src/bt_logger.c · Codeaurora / platform_vendor_qcom-opensource_bluetooth. `https://gitlab.com/Codeaurora/platform_vendor_qcom-opensource_bluetooth/-/blob/db31ce2e09daeb551320b4f9d20e56000123edd0/bt_logger/src/bt_logger.c`, 2017.

[19] 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 *USENIX Security Symposium*, 2020.

[20] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[21] Firmware Panda. Database of Android Stock ROM Firmware for all Android Devices. `https://firmwarepanda.com/`.

[22] G. Gong. Fuzzing Android System Services by Binder Call to Escalate Privilege. BlackHat USA, 2015.

[23] Google Play services. Factory Images for Nexus and Pixel Devices. https://developers.google.com/android/images.

[24] S. Groß, A. Tiwari, and C. Hammer. PIAnalyzer: A Precise Approach for PendingIntent Vulnerability Analysis. In J. Lopez, J. Zhou, and M. Soriano, editors, *European Symposium on Research in Computer Security*, 2018.

[25] G. Hernandez, D. Tian, A. S. Yadav, B. J. Williams, and K. R. B. Butler. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *USENIX Security Symposium*, 2020.

[26] Huawei. Security advisory - Information Disclosure Vulnerability in AEE extension of MTK Platform. https://www.huawei.com/en/psirt/security-advisories/huawei-sa-20170804-01-smartphone-en, Aug 2017.

[27] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru. Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations. In *IEEE International Symposium on Software Reliability Engineering*, 2017.

[28] jcase. [Root] WeakSauce APK - 1.0.1. https://forum.xda-developers.com/t/root-weaksauce-apk-1-0-1.2699089/, Mar 2014.

[29] W. Kai, Z. Yuqing, L. Qixu, and F. Dan. A Fuzzing Test for Dynamic Vulnerability Detection on Android Binder Mechanism. In *IEEE Conference on Communications and Network Security*, 2015.

[30] Y.-T. Lee, W. Enck, H. Chen, H. Vijayakumar, N. Li, Z. Qian, D. Wang, G. Petracca, and T. Jaeger. PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems. In *USENIX Security Symposium*, 2021.

[31] J. Levin. *Android Internals - a Confectioner´s Cookbook*, volume 1, chapter 5, pages 126–129, 137–140, 144. Jonathan Levin, 1st edition, 2015.

[32] man7.org. getsockopt(2) - linux manual page. https://man7.org/linux/man-pages/man2/getsockopt.2.html.

[33] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich. The Android Platform Security Model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3), Apr. 2021.

[34] D. Morse. Samsung releases Professional Audio SDK for Galaxy devices • DJWORX. https://djworx.com/samsung-professional-audio-sdk-galaxy/, Oct 2014.

[35] National Security Agency. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/, 2019.

[36] National Vulnerability Database. CVE-2011-3918. https://nvd.nist.gov/vuln/detail/CVE-2011-3918.

[37] National Vulnerability Database. CVE-2013-4777. https://nvd.nist.gov/vuln/detail/CVE-2013-4777.

[38] National Vulnerability Database. CVE-2013-5933. https://nvd.nist.gov/vuln/detail/CVE-2013-5933.

[39] Pamplemousse. Handle function calls during static analysis in angr. https://blog.xaviermaso.com/2021/02/25/Handle-function-calls-during-static-analysis-with-angr.html, Feb 2021.

[40] A. Possemato, S. Aonzo, D. Balzarotti, and Y. Fratantonio. Trust, but verify: A Longitudinal Analysis of Android OEM Compliance and Customization. In *IEEE Symposium on Security and Privacy*, 2021.

[41] S. Prado. What Differs Android from Other Linux-Based Systems? https://embeddedbits.org/what-differs-android-from-other-linux-based-systems/, 2021.

[42] Qualcomm Technologies, Inc. Qualcomm's CnE Brings "Smarts" to 3g/4g Wi-Fi Seamless Interworking. https://www.qualcomm.com/news/onq/2013/07/02/qualcomms-cne-bringing-smarts-3g4g-wi-fi-seamless-interworking, 2013.

[43] Qualcomm Technologies, Inc. Mobile Security Solutions: Secure Mobile Technology. https://www.qualcomm.com/products/features/mobile-security-solutions, May 2021.

[44] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware. In *IEEE Symposium on Security and Privacy*, 2020.

[45] R. Sasnauskas and J. Regehr. Intent Fuzzer: Crafting Intents of Death. In *Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, 2014.

[46] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao. The Misuse of Android Unix Domain Sockets and Security Implications. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[47] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[48] StatCounter Global Stats. Mobile Vendor Market Share Worldwide. https://gs.statcounter.com/vendor-market-share/mobile/worldwide, May 2021.

[49] D. Tian, G. Hernandez, J. Choi, V. Frost, C. Ruales, K. Butler, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, and M. Grace. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *USENIX Security Symposium*, 2018.

[50] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *USENIX Security Symposium*, Aug. 2015.

[51] D. Yu, G. Yang, G. Meng, X. Gong, X. Zhang, X. Xiang, X. Wang, Y. Jiang, K. Chen, W. Zou, and et al. SEPAL: Towards a Large-scale Analysis of SEAndroid Policy Customization. *Proceedings of the Web Conference*, Apr 2021.

[52] 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 *ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2018.

[53] B. zheng Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge. FANS: Fuzzing Android Native System Services via Automated Interface Analysis. In *USENIX Security Symposium*, 2020.