The Misuse of Android Unix Domain Sockets and Security Implications

Yuru Shao†, Jason Ott*, Yunhan Jack Jia†, Zhiyun Qian*, Z. Morley Mao†
†University of Michigan, *University of California, Riverside
{yurushao, jackjia, zmao}@umich.edu, jott002@ucr.edu, zhiyunq@cs.ucr.edu

ABSTRACT

In this work, we conduct the first systematic study in understanding the security properties of the usage of Unix domain sockets by both Android apps and system daemons as an IPC (Inter-process Communication) mechanism, especially for cross-layer communications between the Java and native layers. We propose a tool called SInspector to expose potential security vulnerabilities in using Unix domain sockets through the process of identifying socket addresses, detecting authentication checks, and performing data flow analysis. Our in-depth analysis revealed some serious vulnerabilities in popular apps and system daemons, such as root privilege escalation and arbitrary file access. Based on our findings, we propose countermeasures and improved practices for utilizing Unix domain sockets on Android.

1. INTRODUCTION

Inter-process communication (IPC) is one of the most fundamental features provided by modern operating systems. IPC makes it possible for different processes to cooperate, enriching the functionalities an operating system can offer to end users. In the context of Android, one of the most popular mobile operating systems to date, to support communications between different apps and interactions between different components of the same app, it provides a set of easy-to-use, Android-specific IPC mechanisms, primarily including Intents, Binder, and Messenger [4, 11]. However, Android IPCs are meanwhile significant attack vectors that can be leveraged to carry out attacks such as confused deputy and man-in-the-middle [23, 15, 17, 19].

While Android relies upon a tailored Linux environment, it still inherits a subset of traditional/native Linux IPCs (which are distinct from Android IPCs), such as signals, Netlink sockets, and Unix domain sockets. In fact, they are heavily utilized by the native layer of the Android runtime. Exposed Linux IPC channels, if not properly protected, could be abused by adversaries to exploit vulnerabilities within privileged system daemons and the kernel. Several vulnerabilities (*e.g.,* CVE-2011-1823, CVE-2011-3918, and CVE-2015-6841) have already been reported. Vendor customizations make things worse, as they expose additional Linux IPC channels: CVE-2013-4777 and CVE-2013-5933. Nevertheless, unlike Android IPCs, the use of Linux IPCs on Android has not yet been systematically studied.

In addition to the Android system, apps also have access to the Linux IPCs implemented within Android. Among them, Unix domain sockets are the only one apps can easily make use of: signals are not capable of carrying data and not suitable for bidirectional communications; Netlink sockets are geared for communications across the kernel space and the user space. The Android software development kit (SDK) provides developers Java APIs for using Unix domain sockets. Meanwhile, Android's native development kit (NDK) also provides native APIs for accessing low-level Linux features, including Unix domain sockets. Unix domain sockets are also known as local sockets, a term which we use interchangeably. They are completely different from the "local socket" in ScreenMilker [25], which refers to a TCP socket used for local IPC instead of network communication.

Many developers use Unix domain sockets in their apps, despite the fact that Google's best practices encourage them to use Android IPCs [4]. The reason being Android IPCs are not suited to support communications between an app's Java and native processes/threads. While there are APIs available in SDK, no such API exists in the native layer [7]. As a result, developers must resort to using Unix domain sockets to realize cross-layer IPC. Furthermore, some developers port existing Linux programs and libraries, which already utilize Unix domain sockets, to the Android platform.

Android IPCs are well documented on the official developer website, replete with training materials and examples. This helps educate developers on best practices and secure implementations. However, there is little documentation about Unix domain sockets, leaving developers to use them as they see fit — this may result in vulnerable implementations. Moreover, using Unix domain sockets securely requires expertise in both Linux's and Android's security models, which developers may not have.

Motivated by the above facts, we undertake the first systematic study focusing on the use of Unix domain sockets on Android. We present *SInspector*, a tool for automatically vetting apps and system daemons with the goal of discovering potential misuse of Unix domain sockets. Given a set of apps, SInspector first identifies ones that use Unix domain sockets based on API signatures and permissions. SInspector then filters out apps that use Unix domain sockets se-

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978297

curely and thus are not vulnerable. We develop several techniques to achieve this, such as socket address analysis and authentication check detection. For system daemons, SInspector collects runtime information to assist static analysis. SInspector reports potentially vulnerable apps and system daemons for manual examination. We also categorize Unix domain socket usage, any security measures employed by existing apps and system daemons, and common mistakes made by developers. From this study, we suggest countermeasures in regard to OS-level changes and secure Unix domain socket IPC for both app and system developers. In this work, we do not consider network sockets, as local IPC is not their common usage.

We find that only 26.8% apps and 15% system daemons in our dataset enforce proper security checks in order to prevent attacks exploiting Unix domain socket channels. All apps using a particular Unix domain socket namespace are vulnerable to at least DoS attacks. We uncover a number of serious vulnerabilities in apps. For example, we are able to gain root privilege by exploiting a popular root management tool, as well as grant/deny any other app's root access, without any user awareness. Moreover, we discover vulnerabilities with customizations on LG phones and daemons implemented by Qualcomm. These vulnerabilities allow us to factory reset the device, toggle the SIM card, and modify system date and time. Attack demos can be found on our project website `https://sites.google.com/site/unixdomainsocketstudy`.

In summary, we make the following contributions:

- We develop SInspector for analyzing apps and system daemons to discover potential vulnerabilities they expose through Unix domain socket channels. We overcome challenges in identifying socket addresses, detecting authentication checks, and performing data flow analysis on native code.

- Using SInspector, we perform the first study of Unix domain sockets on Android, including the categorization of usage, existing security measures being enforced, and common flaws and security implications. We analyze 14,644 apps and 60 system daemons, finding that 45 apps, as well as 9 system daemons, have vulnerabilities, some of which are very serious.

- We conduct an in-depth analysis on vulnerable apps and daemons that fail to properly protect Unix domain socket channels, and suggest countermeasures and better practices for utilizing Unix domain sockets.

2. BACKGROUND

We provide the necessary background to understand the security vulnerabilities in how Android Unix domain sockets are used in apps and system daemons.

2.1 Android Security Model

The Android platform consists of multiple layers. One of Android's design goals is to provide a secure platform so that "[S]ecurity-savvy developers can easily work with and rely on flexible security controls. Developers less familiar with security practices will be protected by safe defaults." [3] Android apps are isolated and run in their own process. They communicate with peer apps through secure, Android-specific IPCs (Binder, Intents, etc). These Android IPC mechanisms, as documented by Google, are the preferred IPC mechanisms as they "allow you to verify the identity of the application connecting to your IPC and set security policy for each IPC mechanism." [4]

However, Unix domain sockets undermine the goals of Android's security philosophy. They are unable to achieve the same guarantees as well as the Android IPCs. In particular, according to our analysis, Android APIs for using Unix domain sockets expose unprotected socket channels by default.

2.2 Unix Domain Sockets

A Unix domain socket is a data communications endpoint for exchanging data between processes executing within the same host operating system. It supports transmission of a reliable stream of bytes (`SOCK_STREAM`, similar to TCP). In addition, it supports ordered and reliable transmission of datagrams (`SOCK_SEQPACKET`), or unordered and unreliable transmission of datagrams (`SOCK_DGRAM`, similar to UDP).

Unix domain sockets differ from Internet sockets in that (1) rather than using an underlying network protocol, all communication occurs entirely within the operating system kernel; and (2) servers listen on addresses in Unix domain socket namespaces, instead of IP addresses with port numbers. Traditionally, there are two Unix domain socket address namespaces, as shown in Table 1.

Table 1: Unix domain socket namespaces.

Namespace	Has socket file	Security enforcement	
		SELinux	File permission
FILESYSTEM	YES	YES	YES
ABSTRACT	NO	YES	N/A

FILESYSTEM. An address in this namespace is associated with a file on the filesystem. When the server binds to an address (pathname), a socket file is automatically created. Socket file permissions are enforced through Linux's discretionary access control (DAC) system. The server must have privilege to create the file with the given pathname, otherwise binding fails. Other processes who want to communicate with the server must have read/write privileges for the socket file. By setting permissions of the socket file properly, the server can prevent unauthorized connections. The Android framework introduces a new namespace called *RESERVED*, which is in essence a sub-namespace of FILESYSTEM. Socket files are located under a particular directory, `/dev/socket/`, reserved for system use.

ABSTRACT. This namespace is completely independent of the filesystem. No file permissions can be applied to sockets under this namespace. In native code, an ABSTRACT socket address is distinguished from a FILESYSTEM socket by setting `sun_path[0]` to a null byte `'\0'`.

The Android framework provides APIs for using Unix domain sockets from both Java code and native code. These APIs use ABSTRACT as the default namespace, unless developers explicitly specify a preferred namespace. All Unix domain socket addresses are publicly accessible from file `/proc/net/unix/`. SELinux supports fine-grained access control for both FILESYSTEM and ABSTRACT sockets, so does SEAndroid. Compared to FILESYSTEM sockets, ABSTRACT sockets are less secure as DAC does not apply. However, they are more reliable; communication over a

Table 2: Types of attacks malware can carry out by exploiting Unix domain sockets.

Role	Prerequisite(s)	Attacks
Malicous Server	1) Start running ahead of the real server 2) Client has no/weak authentication of server	Data Leakage/Injection, DoS
Malicous Client	Server has no/weak authentication of client	Privilege Escalation, Data Leakage/Injection, DoS

FILESYSTEM socket could be interrupted if the socket file is somehow deleted.

2.3 Threat Model and Assumptions

Unix domain sockets are designed for local communications only, which means the client and server processes must be on the same host OS. Therefore, they are inaccessible for remote network-based attackers. Our threat model assumes a malicious app that attempts to exploit exposed Unix domain socket channels is installed on the user device. This is realistic since calling Unix domain socket APIs only requires the `INTERNET` permission, which is so commonly used [2] that the attacker can easily repackage malicious payloads into popular apps and redistribute them. The attacker may also build a standalone exploit app which evades anti-malware products due to its perceived low privilege.

We summarize attacks malware can launch in Table 2. It is able to impersonate either a client or a server to talk to the reciprocal host. A rogue Unix domain socket server could obtain sensitive data from clients or feed clients fake data to impact client functionality. A mock Unix domain socket client could access server data or leverage the server as a confused deputy [24]. In general, we classify a Unix domain socket as vulnerable if the server accepts valid commands through its socket channel without performing any authentication or similarly a client connects to a server without properly authenticating the server. This allows a nefarious user to retrieve sensitive information or access otherwise restricted resources through the Unix domain socket server/-client it communicates with. Moreover, an ABSTRACT address can only be bound to by one thread/process. Apps using ABSTRACT namespace are vulnerable to DoS because their addresses could be occupied by the malware.

3. DESIGN AND IMPLEMENTATION

The goal of SInspector is to examine the use of Unix domain socket in apps and system daemons, and identify those that are most likely vulnerable for validation. In this section, we describe our design and implementation of SInspector.

An ideal solution is to analyze all program paths in a program starting from the point of accepting a Unix domain socket connection, and then identify whether critical functions (end points) can be invoked without encountering any security checks. However, it is not practical for us to define a comprehensive list of end points and use dependencies between entry and end points to reason whether an app is vulnerable. First of all, apps may contain native libraries/executables, in which they make system calls to implement certain functionality, but there is no mapping between Android permissions and Linux system calls. It is imprecise to identify app behaviors based on system calls they make. Second, in our threat model, the malware runs on the same device as the vulnerable app/system daemon to be exploited, thus any data leaked from the target app/system daemon can possibly be a building block for more sophisticated attacks. However, it unknown to us which end points are potentially related to data leakage. More importantly, an incomplete list of end points would result in significant false negatives.

Therefore, to evaluate which apps/system daemons are vulnerable, we choose to conservatively *filter out apps and system daemons that are definitely not vulnerable* (denoted by S_{nv}) — the others are considered to be potentially vulnerable (denoted by S_{pv}) — instead of directly identifying vulnerable apps. We have $S_{pv} = S - S_{nv}$, where S represents the whole set of apps/system daemons.

3.1 Our Approach

Due to different characteristics of apps and system daemons, we adopt different techniques to analyze them. Figure 1 shows the modules and overall analysis steps of SInspector. Each step rules out a subset of apps/system daemons that are not vulnerable.

3.1.1 App Analysis

Given a set of apps, SInspector first employs *API-based Filter* to filter out those not using Unix domain sockets or having insufficient permission to use Unix domain sockets. Then, *Address Analyzer* finds out Unix domain socket address(es) each app uses, and discards apps whose addresses are under protection. They are not vulnerable because proper socket file permissions are able to prevent unauthorized accesses to a filesystem-based Unix domain socket channel. Next, the apps left are further examined by *Authentication Detector*. It detects and categorizes authentication mechanisms apps implement. Those adopting strong checks are considered to be not vulnerable. After that, *Reachability Analyzer* checks whether the vulnerable code that uses Unix domain socket will be executed or not at runtime. If not, that code is not reachable and will never be triggered, thus the app is not vulnerable. It ends up with a relatively small set of apps that are potentially vulnerable. Manual efforts are finally required to confirm the existence of vulnerabilities.

API-based Filter. This module filters out apps that are not in our analysis scope. For each app, it checks (1) Android permissions the app declares, (2) Java APIs the app calls, and (3) Linux system calls if the app has native code. Since using Unix domain sockets requires the `INTERNET` permission, apps without this permission are surely not vulnerable, neither are apps that do not invoke related APIs or system calls. APIs called through Java reflection are currently not considered, because (1) all socket APIs are available in Android SDK, unlike some private or hidden APIs which can only be called via Java reflection; and (2) Unix domain sockets just require a common, non-dangerous permission and therefore apps have little intention to hide the relevant logic.

Address Analyzer. This module identifies socket addresses each app uses and determines if their corresponding Unix domain socket channels are protected. Dalvik byte

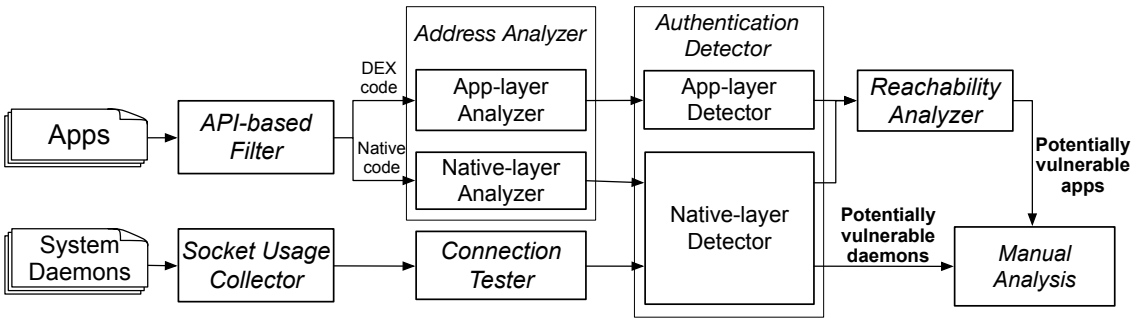

Figure 1: Overview of our approach to identifying potentially vulnerable apps and system daemons.

code and native code are analyzed by Address Analyzer's two submodules, *App-layer Analyzer* and *Native-layer Analyzer*, respectively.

Being aware of Unix domain socket address(es) an app connects to and/or listens on has two benefits. First, we can leverage addresses to determine if both client logic and server logic present in the same app. Usually it is much easier to craft server exploits by replaying client behaviors, and vice versa. Second, different apps may use common libraries that utilize Unix domain sockets to implement certain functionality. We can take advantage of addresses to better group apps according to the libraries they use, because of the fact that apps using the same library typically have the same Unix socket address (or address structure). This is more reliable than identifying libraries merely based on package names and class names, as package names and class names could be easily obfuscated by tools like ProGuard [9]. Though code similarity comparison techniques are also capable of recognizing libraries used across different apps, they are usually heavyweight.

Besides identifying addresses, Address Analyzer also evaluates whether the socket channel on an address is secure or not. As we have mentioned in §2, when using FILESYSTEM addresses, Unix domain socket servers are able to restrict client accesses by setting proper file permissions for socket files they listen on. A socket file satisfying the following conditions has proper permissions, and therefore the app using it is considered not vulnerable. First, it is located in the app's private data directory, *i.e.,* `/data/data/app.pkg.name/`. By default socket files created under this directory can only be accessed by the app itself. Second, there is no other operation altering the socket file's permissions to publicly accessible. The app, as the socket file's owner, has the privilege to change its permissions to whatever it wants. All file operations that possibly change the socket file's permissions need to be examined.

Authentication Detector. The OS allows both the client and the server to get their peers' identity information (*i.e.,* peer credentials) once a Unix domain socket connection is established. This module detects and categorizes authentication checks built on peer credentials. It also consists of two submodules for processing non-native and native code separately. Peer credentials are only available for Unix domain sockets. In our threat model, they are absolutely reliable because they are guaranteed by the kernel and therefore cannot be spoofed or altered by any non-root process in the user space. In Java code, apps call Android SDK API `LocalSocket.getPeerCredentails()` to get a socket peer's credentials, containing three fields: PID, UID, and GID. While in native code, the system call `getsockopt` is used to obtain the same information. Based on UID, GID and PID, servers and clients can implement various types of peer authentication mechanisms. Authentication Detector keeps track of the propagation of peer credentials in code, detects checks built upon the credentials, and categorizes them according to the particular credential they depend on. Peer authentication checks derived from UID and GID are considered to be strong, as UID and GID are assigned by the OS and cannot be spoofed. However, authentications based on PID are relatively weak. Further analysis is unnecessary for apps employing strong checks.

Reachability Analyzer. The presence of Unix domain socket APIs in code does not necessarily mean the app actually uses Unix domain sockets at runtime. It is possible that the app just imports a library that offers functionality implemented with Unix domain sockets, but that part of code is never executed. To filter out such apps, Reachability Analyzer collects all possible entry points of an app, from which it builds an inter-component control flow graph. If Unix domain socket code cannot be reached from either of the entry points, we believe the code will not be reached at runtime, thus the app is considered not vulnerable.

3.1.2 System Daemon Analysis

Several obstacles make pure static analysis of system daemons infeasible. First, given a factory image that contains all system files, it is difficult to extract all required data from it due to the fact that vendors develop their own file formats and there is no universal tool to unpack factory images. Second, different from apps, system daemons' Unix domain socket channels are usually enforced with specific SEAndroid policies made by Google or vendors. In this case, evaluating the security of a Unix domain socket channel becomes more complicated, especially for the FILESYSTEM namespace, because it is determined by both SEAndroid and socket file permissions.

However, system daemons are suitable for dynamic analysis without worrying about potential code coverage issues. They start automatically, serve as Unix domain socket servers waiting for client connections, and provide no user interface. It is reasonable to assume that their server logics are always running instead of being started on demand. Therefore, instead of employing API-based Filter and Address Analyzer, SInspector collects runtime information to find out system daemons using Unix domain socket with *Socket Usage Collector*, then test all socket channels dae-

mons expose with *Connection Tester*, to see which ones are accessible for an unprivileged app. The native layer Authentication Detector is reused for detecting and categorizing checks inside system daemons.

Socket Usage Collector. It is impossible for us to exploit vulnerable client logics implemented inside system daemons. One prerequisite of attacking client is being able to start running before the real server. In our threat model, however, the third-party app with only `INTERNET` permission can never run ahead of a system daemon, which is started by the `init` process even before the Android runtime is initialized. Socket Usage Collector gathers runtime information of each Unix domain socket, including address, the process that listens on the address, protocol type (DGRAM, STREM, or SEQAPCKET), and corresponding system daemon.

Connection Tester. According to socket channel information collected, Connection Tester attempts to connect to them one by one, acting like a client running as a third-party app with `INTERNET` permission. If a socket channel is enforced by either file permissions or SEAndroid policies, connection will be denied because of insufficient privilege. A system daemon is not vulnerable if all its socket channels are well protected.

3.1.3 Manual Analysis

For apps and system daemons that are most likely to be vulnerable, manual reverse engineering efforts are required to investigate the existence of vulnerabilities. Various tools are helpful for statically and dynamically reversing apps, *e.g.,* JEB [8], the Xposed framework [12], and IDA Pro. The effort needed for validating vulnerable code is supposed to be minimal, although writing workable exploits may take longer. Message formats (or called protocols) apps and system daemons use could be quite ad-hoc. Reverse engineering efforts largely depend on the complexity of implementation. In order to reduce human efforts, we could integrate protocol reversing techniques proposed in prior work [16, 18, 26] into SInspector in the future.

3.2 Implementation

We implement SInspector based on two cutting-edge tools, Amandroid [31] and IDA Pro. Both of them offer great extensibility and are friendly to plugin development. We take advantage of Amandroid to build inter-procedural control flow graph (ICFG), inter-procedural data flow graph (IDFG), and data dependence graph (DDG) from apps' non-native part for performing app-layer analysis, and leverage IDA Pro's disassembler and control flow analysis to build data flow analysis on native code, including apps' ELF libraries/executables and system daemons. SInspector only supports 32-bit ARM binaries for now, considering that the majority of Android devices are equipped with 32-bit ARM architecture processors.

Analyzing Apps. API-based Filter extracts `AndroidManifest.xml`, decodes it, and looks for the `INTERNET` permission. App code written in Java is compiled into one or more DEX files, in which all invoked APIs are visible. Native binaries are in ELF format. IDA Pro is able to identify direct system calls represented as constant relative addresses embedded in the instructions. However, it does not resolve indirect call targets that are stored in registers. More specifically, binaries can use the `SVC` instruction to do system calls, by specifying a system call number in register `R7` and then executing `SVC 0`. We extract the mapping between system call numbers and system call names from `arch/arm/include/asm/unistd.h` found in Android kernel 3.14, and identify all indirect system calls by inspecting `R7`'s values before each `SVC 0` instruction.

The app-layer of Address Analyzer and Authentication Detector are implemented on top of Amandroid. The server logic and the client logic are analyzed separately. We first locate the method in which Unix domain socket server/-client is initialized, and create a customized entry point to it, then invoke Amandroid to build ICFG, IDFG and DDG from the entry point. In Java code, Unix domain socket address is represented by the `LocalSocketAddress` class, whose constructors accept an address string as the first parameter. We look at construction sites of `LocalSocketAddress` objects. In some cases, constant strings are used. In other cases where an address is built from package name, random integer, *etc.*, we track its construction of procedure by querying dependencies on DDG. Such an example is shown in Figure 2, in which we need to apply data flow analysis to extract the address as [*"com.qihoo.socket"* + *System.currentTimeMillis*()%65535]. This allows us to group apps that share the same socket address or have the same address construction procedure.

```
public static String getAddr() {
  return String.format("com.qihoo.socket%x",
    Long.valueOf(System.currentTimeMillis() & 65535));
}

protected void b(…) {
  …
  String addr = getAddr();
  this.serverSock = new LocalServerSocket(addr);
  …
}
```

Figure 2: A dynamically constructed socket address case.

The app-layer Authentication Detector finds paths on ICFG from `LocalServerSocket.accept()` (for server) and `LocalSocket.connect()` (for client) to `LocalSocket.getInputStream()` or `LocalSocket.getOutputStream()`. If we find that `LocalSocket.getPeerCredentials()` is called along the paths, and there is control dependency between either `getInputStream()`/`getOutputStream()` and `getPeerCredentials()`, authentication happens. In order to categorize authentication checks, we look at which fields (UID, GID or PID) are retrieved. We also define methods in `Context` and `PackageManager` that take UID, GID, or PID as sinks, and run taint analysis to track propagation paths. As mentioned in §3.1, checks relying on UID and GID are considered strong, while others are weak.

The native-layer Address Analyzer leverages intra-procedural control flow graph (CFG) generated by IDA Pro. Each basic block consists of a series of ARM assembly code disassembled by IDA Pro's state-of-the-art disassembling engine. We perform intra-procedural data flow analysis on the CFG, following the classical static analysis approach [28]. Computing data flow at the assembly level is complicated, since we have to take into consideration both registers and the function stack. Unfortunately there does not exist any robust tools that can perform data flow analysis on ARM binaries. ARM is a load-store architecture, and no instructions directly operate on values in memory. This means val-

ues must be loaded into registers in order to operate upon them. Therefore, we need to carefully handle all commonly used instructions that operate on registers and memory, especially load and store (pseudo) instructions. We examine the second argument of system calls `bind()` and `connect()`, which is an address pointing to the `sockadd_un` structure. Unix domain socket string is copied to the `sun_path` field, 2 byte off the start of `sockadd_un`. The first byte of `sun_path` indicates address namespace.

The native-layer Authentication Detector also performs intra-procedural data flow analysis. `getsockopt` has five parameters in total. Among them, the third one (option name) and the fourth one (option value pointer) are crucial. When option name is an integer equal to 17 (macro `SO_PEERCRED`), the option value will be populated by peer credentials, a structure consisting of three 4-byte integers: PID, UID, and GID. In other words, suppose option value's address is `A`, PID, UID, and GID will locate at addresses `A`, `A+4`, and `A+8`, respectively. When `getsockopt` is called, we inspect option name and record option value's address on the stack `A`. After that, functions that access values at `A`, `A+4`, or `A+8` are considered as checks.

Analyzing System Daemons. Socket Usage Collector calls a command line tool `netstat` to get interested socket information. Note that the default `netstat` shipped with Android has very limited capability. We choose to install `busybox`, which provides a much more powerful `netstat` applet. Root access of the Android device is required, otherwise `netstat` will not be able to find out the process that listens on a particular socket address. We build Connection Tester into a third-party app that requests only `INTERNET` permission. Native-layer Authentication Detector is reused for analyzing system daemons.

3.3 Limitations

One limitation of SInspector is that we have to rely on human efforts to generate exploits. Even though we can find out apps and system daemons that are highly likely to be vulnerable, we are not able to automatically craft exploits to finally validate vulnerabilities. SInspector may have false positives, because of our conservative strategies for filtering out insusceptible apps and system daemons. The native-layer intra-procedural data flow analysis is likely to miss data flows across different functions.

We may also have false negatives: (1) we cannot handle dynamically loaded code; and (2) native executables/libraries might be packed or encrypted. They could introduce uncaught control and data flows.

4. RESULTS

We evaluate SInspector with a total number of 14,644 up-to-date Google Play apps crawled by ourselves in mid-April 2016, including (approximately) top 340 from all 44 categories. Google has imposed restrictions to ensure that apps can only be downloaded through the Google Play app, which makes it difficult for us to obtain APK files. To tackle this, we crawl meta data of apps (*e.g.,* package name, version name) from Google Play and download corresponding APK files from ApkPure [5], a mirror of Google Play that allows free downloading.

We also use three phones to evaluate SInspector: (1) LG G3 running Android 4.4.4, (2) Samsung Galaxy S4 running Android 5.0.1, and (3) LG Nexus 4 running 5.1.1. All of them are updated to the latest firmware and rooted. Most of recently released Android phones either equip with 64-bit ARM processors or cannot be rooted. They are not suitable for our experiments because SInspector's dynamic analysis requires root access and the static data flow analysis can only handle 32-bit ARM binaries.

4.1 Overview

Table 3 shows the overall statistics on Unix domain socket usage among apps and system daemons. App data are from API-based Filter and daemon data come from Socket Usage Collector. Among 14,644 apps, 3,734 (25.5%) have Unix domain socket related APIs or system calls in code, and the majority of them (3,689) use ABSTRACT addresses, while only a few use FILESYSTEM and RESERVED addresses.

Different from apps, most of system daemons use RESERVED addresses. Compared to Nexus 4 running non-customized Android, LG G3 and Galaxy S4 have more system daemons and heavier usage of ABSTRACT addresses. This fact clearly shows that vendor customizations inevitably expose more attack vectors.

Table 3: Numbers of apps/system daemons that use Unix domain sockets. The sum of numbers in each address namespace may be greater than the total number, as one app/system daemon could use more than one namespaces.

	# Apps	# Daemons		
		LG G3	Galaxy S4	Nexus 4
ABSTRACT	3,689	5	8	2
FILESYSTEM	36	4	5	2
RESERVED	20	13	17	11
Total	3,734	20	27	13

4.1.1 Libraries

We summarize identified libraries utilizing Unix domain sockets in Table 5. "Singleton" and "Global lock" in the Usage column will be described later in §4.2. We observe that 3,406 apps use an outdated Google Mobile Services (GMS) library alone and exclude them. The outdated GMS library is potentially vulnerable to DoS and data injection attacks. The latest GMS library has completely discarded Unix domain sockets, which implies that Google may have been aware of potential problems of using Unix domain sockets. Except Amazon Whisperlink and OpenVPN, all other libraries use the ABSTRACT namespace, making them all vulnerable to DoS.

4.1.2 Tool Effectiveness and Performance

Besides apps using common libraries listed in Table 5, SInspector found 73 potentially vulnerable apps having no authentication or weak authentications. Table 4 summarizes analysis effectiveness. After reachability analysis, SInspector finally reported 67 apps that are most likely to be vulnerable. We manually looked at all 67 apps and confirmed that 45 are indeed vulnerable. SInspector reported 12 potentially vulnerable system daemons. After manual examination, we confirmed 9 of them are truly vulnerable. We present a case study of most critical vulnerabilities in §5.

All experiments are done on a machine with 3.26GHz × 8 Core i7 and 16GB of memory. The most compute-intensive module of app analysis is Reachability Analyzer. Depending

Table 5: Libraries that use Unix domain socket. ABS and FS under "Namespace" are short for ABSTRACT and FILESYSTEM. DI and DL in the last column stand for data injection and data leakage.

Library	# Apps (reachable)	Usage	Namespace	Auth	Susceptible attack(s)
Baidu Push	9 (9)	Singleton	ABS	N/A	DoS
Tencent XG	11 (11)	Singleton	ABS	N/A	DoS
Umeng Message	17 (17)	Singleton	ABS	N/A	DoS
Facebook SocketLock	13 (13)	Global lock	ABS	N/A	DoS
Yandex Metrica	95 (95)	Global lock	ABS	N/A	DoS
Facebook Stetho	97 (97)	Debugging interface	ABS	Permission	DoS
Sony Liveware	8 (5)	Data transfer	ABS	None	DoS, DI, DL
Samsung SDK	12 (10)	Data transfer	ABS	None	DoS, DI, DL
QT5	10 (10)	Debugging interface	ABS	None	DoS, DI, DL
Clean Master	9 (9)	Data transfer	ABS	None	DoS, DI, DL
Amazon Whisperlink	11 (7)	Data transfer	FS	None	Not vulnerable
OpenVPN	7 (4)	Cmd & control	FS	None	Not vulnerable

Table 4: Results summary.

	Potentially Vulnerable	True Positive	False Positive	Precision
Apps	67	45	22	67.2%
LG G3	6	4	2	66.7%
Galaxy S4	5	4	1	80%
Nexus 4	1	1	0	100%

on the numbers of bytecode instructions of apps, Reachability Analyzer could take a few minutes to more than one hour. Other modules are pretty fast. The average time for analyzing one app is 2,502 seconds. For system daemon analysis, IDA Pro's disassembling process took a few seconds to a few minutes, the average time for analyzing a system daemon is 39 seconds.

4.2 Unix Domain Socket Usage

Unix domain sockets provide a means to perform IPC, but it turns out the usage in the wild is not limited to IPC. According to our experience in inspecting potentially vulnerable apps SInspector reported, we extract code patterns for categorizing Unix domain socket usage and summarize them in Table 6. We observe that Unix domain sockets are widely used by apps to implement global locks and singleton, as well as to implement watchdogs.

4.2.1 *Inter-Process Communication*

Not surprisingly, the prominent usage of Unix domain sockets is performing IPC. Apps are free to implement their own protocols for client/server communication.

However, we do find a very unique use of Unix domain socket as an IPC mechanism. A few video recording apps leverage Unix domain sockets to realize real-time media streaming, a feature that Android's media recording APIs do not support. Developers came up with a workaround, which takes advantage of an existing media recording API `setOutputFile(fd)` that outputs camera and microphone data stream to a file descriptor. After a Unix domain socket connection is established, the client passes its output file descriptor to this API so that the server can read real-time camera/microphone output. In this way, media output is converted to a stream that can be further processed in real time, *e.g.,* to perform live streaming.

4.2.2 *Realizing Singleton*

An ABSTRACT socket address can only be bound on by one Unix domain socket server instance. Once an address has been taken, another server that attempts to bind on it would fail. This feature is widely exploited to ensure that certain code will not be executed more than once. In fact, the `PhoneFactory` class in AOSP "use UNIX domain socket to prevent subsequent initialization" of the `Phone` instance, as Figure 3 shows.

```
105 try {
106   // use UNIX domain socket to
107   // prevent subsequent initialization
108   new LocalServerSocket("com.android.internal.telephony");
109 } catch (java.io.IOException ex) {
110   hasException = true;
111 }
```

Figure 3: `com.android.internal.telephony.PhoneFactory` uses a Unix domain socket for locking. Code excerpted from AOSP `6.0.1_r10`.

Baidu Push, Tencent XG, and Umeng Message are three top message push service providers in China. Due to the state-level blocking of Google services, Google Cloud Messaging (GCM) is not accessible. Therefore, apps targeting on China market have to choose other push services. It is likely that multiple apps integrated the same push service library co-exist on the same device. That would be less power-efficient if they each run their own push service. They choose to share one push service instance across multiple apps and realize that with a Unix domain socket.

4.2.3 *Implementing Global Lock*

This use case also takes advantage of the feature that ABSTRACT addresses are used exclusively. There is demand on global locks because some resources cannot be used by two different processes/threads simultaneously, or certain operations should be serialized instead of parallelized. However, Android itself does not provide global locks shared between different apps. Facebook apps all have a DEX optimization service. They will not do optimization before successfully acquiring a global lock implemented with a Unix domain socket. This ensures that only one optimization task

Table 6: Code patterns for categorizing Unix domain socket usage.

Usage	Key APIs	Code Pattern	# Apps

IPC	`LocalSocketServer.<init>()`
	`LocalSocketServer.accept()`
	`LocalSocket.connect()`
	`LocalSocket.getInputStream()`
	`LocalSocket.getOutputStream()`
		Unix domain socket server/client reads data from (or write data to) the other end.	193

Singleon/ Global Lock	`LocalServerSocket.<init>()` `LocalSocket.bind()`	Server has no reading/writing operations after binding to an address.	165
Watchdog	`LocalSocket.connect()` `LocalSocket.getInputStream()`	Client connects to server and then blocks at reading. Server also blocks at reading after accepting client connection.	33

runs in background, and helps reduce negative impact on user experience.

4.2.4 Implementing Watchdog

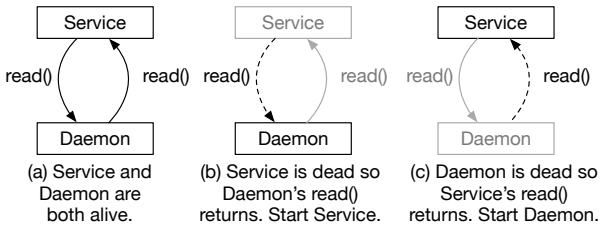

Figure 4: The Kaspersky app's service and daemon monitor each other through a Unix domain socket.

Some apps have important services that are expected to always run in background. Such "immortal" services are against Android's memory management philosophy, and therefore developers have to to find a workaround to automatically restart them, in case they are somehow terminated. They implement a watchdog mechanism leveraging Unix domain sockets. For example, the Kaspersky Security app starts a native daemon in a service. The daemon and the service monitor each other mutually, through a Unix domain socket channel. If one is died, the other will gets notified and restart it immediately, as Figure 4 depicts.

4.3 Peer Authentication

We refine the categorization made by SInspector's Authentication Detector module, and classify peer authentication checks into four types: UID/GID checks, process name checks, user name checks, and permission checks. Table 7 shows the numbers of apps and system daemons adopting each type of checks. Apps and daemons tend to use different types of authentication checks. Apps only adopt UID/GID checks and permission checks, while system daemons use all checks except permission checks. One possible reason is in different layers the information apps/system daemons can obtain differs. In app layer, apps can easily get the peer app's permissions with its UID. However, there are no APIs for getting the peer's process name or user name. In native layer, process name and user name can be easily obtained. But due to the lack of Android runtime context, it is infeasible to query the peer's permissions. Only 9 of 60 (15%) daemons employ strong checks, meaning that their security heavily rely on the correctness of SEAndroid policies and file access permissions.

Table 7: Statistics on peer authenication checks.

	UID/GID	Process name	User name	Permission
#Apps	20	0	0	97
#Daemons	7	3	2	0

Process Name Checks. In native layer, getting process name with its PID is done by reading `/proc/PID/cmdline` or `/proc/PID/comm` on the proc filesystem (procfs). Process name checks compare the peer's process name with predefined process name(s). By default, the process name of an Android app is its package name. Therefore, the content of the two proc files of an app process is actually the app's package name. Interestingly, we find that apps are able to modify their own process names at runtime, by calling a hidden method `Process.setArgV0(String s)` through Java reflection. This method is supposed to be used by the system (labeled with `@hide` in source code), but it requires no permissions. This hidden method makes all process name checks meaningless, as malicious apps can always change their process names to legitimate ones so that they can bypass checks and send messages to the victim. For example, the system daemon `cnd` on LG G3 and Galaxy S4 is used for managing Qualcomm connectivity engine [10]. It accepts requests from clients through a Unix domain socket and checks if the client's process name is "android.browser". Requests from other clients are not legitimate and will be discarded. By changing process name to "android.browser", any app can send legitimate requests to `cnd` effortlessly.

UID/GID Checks. Android reserves UIDs less than 10,000 for privileged users. For instance, the user `system` has 1,000 as both UID and GID. Normally, each app has its own UID and GID, but apps from the same developer could share the same UID and GID. These checks are handy when one wants to allow only privileged users or particular apps to communicate with it. UID/GID checks efficiently prevent unauthorized peers, as UID and GID can never be spoofed or modified. For example, the Android Wear app has a service called AdbHubService, which is used for remote debugging. It starts a Unix domain socket server accepting debugging commands from ADB shell. Only commands coming from `root` and `system` are allowed, by checking if a client's UID is equal to 0 or 2,000.

User Name Checks. These checks are similar to UID/GID checks, since each user also has its unique user name that cannot be spoofed or modified. They also effectively authenticate the peer's identity. Samsung Galaxy S4's RIL daemon, `rild`, checks client user name. A list of names

of privileged users are hardcoded in the binary, *e.g.,* media, radio. User name checks might be better than UID/GID checks because the same user may have different UID/GID on different devices due to vendor customization.

Permission Checks. These checks enforce that only apps with specific permissions can access the Unix domain socket channel. In app layer, apps can call several APIs in the `Context` class to check another app's permissions. The Facebook Stetho library checks if the peer has the `DUMP` permission, a system permission that can only be acquired by system apps. It first obtains UID and PID from peer credentials, then calls `Context.checkPermission(permission, pid, uid)` to do permission checking

Token-based Checks. Besides aforementioned peer authentication checks, we observe two apps adopt token-based checks. The server and the client first securely share a small chunk of data (called *token*). The server compares the token of the incoming client with its own copy so that only clients having the right token can talk to it. This type of checks, assuming the token is shared in secure ways, can effectively prevent unauthorized accesses. We find two apps employing two different methods to share tokens between the server and the client. The first one, Helium Backup, broadcasts the token on the server side. The broadcast is protected by a developer-defined permission, and therefore other apps without the required permission cannot receive the token. The second one, OS Monitor, stores its token in a private file. Since the server and the client are both created by the app itself, they have privileges to read the private file and extract the token. SInspector currently cannot identify such checks. As a result, these apps reported as potentially vulnerable are actually false positives.

5. CASE STUDY

By examining the output of SInspector, we successfully discovered several high-severity zero-day vulnerabilities affecting popular apps installed by hundreds of millions of users, widely used third-party libraries, and system daemons having root privileges. These vulnerabilities can be exploited to (1) grant root access to any apps, giving the attacker entire control of the device, (2) read and write arbitrary files, allowing the attacker to steal user privacy and modify system settings, (3) factory reset the victim device, causing permanent data loss, and (4) change system date and time, resulting in denial of service. Attack demos and more details are available on our project website `https://sites.google.com/site/unixdomainsocketstudy`.

5.1 Applications

5.1.1 Data Injection in a Rooting Tool

As rooting gaining popular in the Android community, many one-click rooting tools become available [33], which allow users to gain root access very easily. One major rooting tool, which claims to be able to root 103,790 different models (as of May, 2016), support a wide range of devices running Android 2.3 Gingerbread and above up to Android 6.0 Marshmallow. As well as rooting, the tool also serves as a root access management portal, through which users can grant or deny apps' root requests.

Once a device is successfully rooted, the rooting app installs a command line tool, `su`, to the system partition `/system/bin/su`. Apps then request root access by executing `su`,

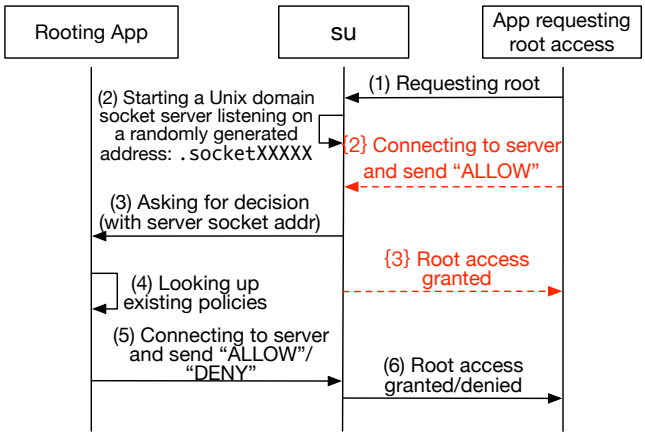

Figure 5: Vulnerability illustration. The normal root request procedure consists of steps (1)-(6) with solid arrow lines. By injecting "ALLOW" any app can get root access regardless what the user's actual decision is, shown as steps (1){2}{3}.

who starts a Unix domain socket server waiting for the rooting app to send back user decision. The rooting app looks up existing policies. If no corresponding policy exists, it pops up a dialog that asks the user to make decision. Figure 5 illustrates the whole process.

However, the FILESYSTEM-based socket channel is publicly accessible as its file permissions are set to `rwxrwxrwx`, and there is no client authentication in `su`. As a result, any app can inject arbitrary decisions before the rooting app sends out the real decision to `su`. This allows a malicious app to grant or deny root access of any other apps, as well as grant itself root privileges in order to take full control of the device. We reported this vulnerability to the developers and they rated it as the most severe security bug in their product to date. They fixed the vulnerability and released a new version in 24 hours.

5.1.2 Privilege Escalation in ES File Explorer

ES File Explorer is a very popular file management app on Android, accumulating over 300 million installs [6]. To perform file operations that Java layer APIs cannot efficiently support, the app starts a native process and executes `libestool2.so`[1], which creates a Unix domain socket server listening on an ABSTRACT address, `@/data/data/com.estrongs.android.pop/files/comm/tool_port`. Moreover, if the device is rooted and the user chooses to run ES File Explorer in root mode, it starts another `libestools2.so` process with root privileges, listening on another ABSTRACT address, `@/data/data/com.estrongs.android.pop/files/comm/su_port`. Certain low-level operations, such as modifying file permissions and changing file status and ownership are sent to these two native processes to execute.

Since there is no client authentication on the server side (*i.e.,* `libestool2.so`), any app can send them commands to run. We were able to read any app's private files and protected system files by exploiting this vulnerability, after successfully reversing the communication protocol used by the ES File Explorer app and its native processes. This

[1]This binary looks like a shared library from its name, but it is essentially an ELF executable.

vulnerability was fixed two months after we first reported it to the developers.

5.1.3 DoS VPN Apps

Multiple OpenVPN clients for Android are available. *OpenVPN for Android* is an open source client that targets at advanced users and offers many settings and the ability to import profiles from files and to configure/change profiles inside the app. The client is an ELF executable ported from the community version of OpenVPN for Linux.

OpenVPN management interface allows OpenVPN clients to be administratively controlled from an external program via a TCP or Unix domain socket. Quite a few of apps making use of OpenVPN for Android utilize Unix domain sockets to communicate with the management app. However, some of them fail to set file permissions correctly for the socket file. OpenVPN supports various client authentication mechanisms. Surprisingly, none of these apps adopt any client authentication. Consequently, an adversary can establish connection to the management interface and then control the OpenVPN client, causing deny-of-service at least.

5.2 System Daemons

5.2.1 LG AT daemon

The privileged *AT Daemon*, `/system/bin/atd`, on (at least) the LG G3 is vulnerable, which allows any app with only the `INTERNET` permission to factory reset the phone, toggle the SIM card, and more, causing permanent data loss and denial of service. `atd` is a proprietary daemon developed by LG. It starts a Unix domain socket server that performs no client authentication, listening on socket file `/dev/socket/atd`, whose permissions are not correctly configured (*i.e.,* `srw-rw---- system inet`). The permission configuration means all users in the `inet` Linux group can read and write this socket file. Android apps having the `INTERNET` permission all belong to the `inet` group. As a result, they are able to read and write this socket file so that they can talk to the AT daemon through this Unix domain socket channel. Commands from any apps, if in the right format, will be processed by the daemon.

By reversing the message format `atd` accepts, we successfully crafted commands that can instruct `atd` to perform factory reset, wiping all user data and toggle the SIM card. In fact, `atd` accepts a large set of commands (only a subset were successfully reversed); reverse engineering the protocol allows us to send arbitrary SMS requests, make phone calls, get user's geographic location, *etc.*. This vulnerability has been assigned CVE-2016-3360.

5.2.2 Qualcomm Time Daemon

We first found that a LG G3 daemon `/system/bin/time_daemon` opens a Unix domain socket server listening on an ABSTRACT address `@time_genoff`. This daemon verifies the client's identity. However, verification is weak and can be easily bypassed — it only checks whether the process name of the client is a constant string "comm.timeservice".

This vulnerability allows any app with the `INTERNET` permission to change the system date and time, affording attackers to DoS services relying on exact system date and time, e.g., validating server certificate. `/system/bin/time_daemon` is developed by Qualcomm, therefore other Android phones using Qualcomm time daemon are also vulnerable. This vulnerability has been reported and was assigned CVE-2016-3683.

5.2.3 Bluedroid

The Android Bluetooth stack implementation is call bluedroid, which exposes a Unix domain socket channel for controlling the A2DP protocol [1]. The ABSTRACT address, `@/data/misc/bluedroid/.a2dp_ctrl`, is expected to be enforced by SEAndroid. To our surprise, we are able to connect to the server through this address and send control commands to it on a Nexus 4. We are able to control the audio playing on a peripheral device connected to the phone through Bluetooth. Though the LG G3 and the Galaxy S4 also expose the same channel, accesses from third-party apps always fail at connecting stage due to insufficient permission. This case suggests that vendors may have made some security improvements despite their tendency to introduce vulnerabilities [34].

6. COUNTERMEASURE DISCUSSION

As our study suggests, the misuse of Unix domain sockets on Android has resulted in severe vulnerabilities. We discuss possible countermeasures to minimize the problem from two aspects: (1) OS-level mitigations and (2) better approaches to implementing secure IPC that utilizes Unix domain sockets.

6.1 OS-level Solutions

Changing the default namespace. For now, Unix domain socket channels created by apps use the ABSTRACT namespace by default. Due to the lack of DAC, socket channels based on ABSTRACT addresses are less secure than those based on FILESYSTEM addresses. Therefore, an intuitive mitigation is to change the default namespace from ABSTRACT to FILESYSTEM; or more radically, disable the use of ABSTRACT namespace.

More fine-grained SEAndroid policies and domain assignment. In the current SEAndroid model, all third-party apps, although having individual UIDs and GIDs, are assigned the same *domain* label, *i.e.,* `untrusted_app`. Unix domain sockets accesses between third-party apps are not enforceable by SEAndroid because domain-level policies cannot tell one third-party app from another.

Therefore, we need to assign different *domain* labels to different third-party apps so that more fine-grained policies can be made to regulate Unix domain socket accesses. Nevertheless, this could introduce new problems: pre-defined policies would not be able to cover apps, and making fixed policies editable at runtime may open new attack vectors. Moreover, it would be untenable to define policies for every app; each user may install any number of apps.

6.2 Secure IPC on Unix Domain Sockets

We demonstrate three scenarios where apps and system daemons require Unix domain sockets for IPC and discuss possible solutions to their security problems.

A privileged system daemon exposes its functionality to apps. A system daemon may need to provide diverse functionality to apps that have different privileges. For example, the LG AT daemon may want to expose the capability of doing factory reset to only system apps, and allow apps with location permissions to get the user's GPS coordinates. To achieve this, system daemons will have to

enforce app permissions themselves. Unfortunately, the lack of Android runtime context in system daemons precludes daemons from easily obtaining the app's permission(s).

Figure 6 demonstrates the proposed solution. The goal being to delegate peer authentication to the existing Android security model. Instead of letting apps and daemons communicate directly through a Unix domain socket, a system service acts as an intermediary between the two. This new system service runs as the `system` user with UID 1000, thus can be easily authenticated by the daemon. Apps talk to this system service through Android Binder and their permissions are validated by the system service. In this way, daemon functionality is indirectly exposed to apps with the help of a system service.

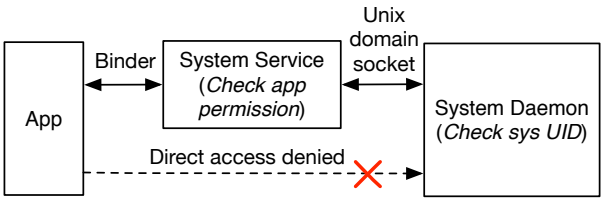

Figure 6: A secure way to expose system daemon functionality to apps. A system service is added between apps and the system daemon.

An app consisting of both Java and native code performs cross-layer IPC. Apps having native executables need an intra-application, cross-layer IPC. An app creates a native process to run its executable, and uses Unix domain sockets to communicate with the native process from its non-native part. In this case, executables still have the same UID as their owner apps. Therefore, it is convenient to check UID on both client and server sides.

An app exposes interfaces to other apps. Android-specific IPCs such as Intents are expected to be used for inter-application communications. However, apps have to choose Unix domain sockets for cross-layer IPCs. We propose a token-based mechanism inspired by Helium described in §4.3, as Figure 7 illustrates. The client app first sends a broadcast to the server app to request a communication token. The server responds by asking the user to allow or deny the incoming request. If the user allows, the server app generates a one-time token for that particular client and returns the token. After that, the client connects to the server with its token and a Unix domain socket connection will be established. Note that the token is not meaningful to anyone else. Even if it was stolen, the attacker would not be able to use it to talk to the server app.

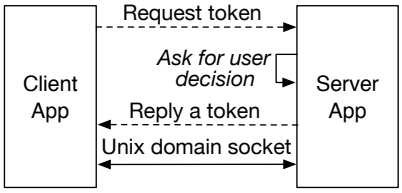

Figure 7: Token-based secure Unix domain socket IPC. Dotted arrow lines stand for permission-protected broadcasts.

7. RELATED WORK

As the community continues to explore and understand Android and its ecosystem, novel attacks and innovative ways of uncovering vulnerabilities are being developed. Many of the existing works in Android security leverage static and/or dynamic analysis of Android apps and framework. By comparing our work with that of others we distinguish and demonstrate how our work contributes to Android security research.

Android IPC and framework vulnerabilities. The Android IPC mechanisms, *e.g.,* Binder, Messenger, and Intents, have been thoroughly studied [17, 27, 20, 14, 31, 22]. These works aim to exploit the IPC mechanisms in order to disclose sensitive information such as SMS messages, call history, and GPS data [30]. For example, Chin *et. al.* [17] examined Android application interaction and identified security risks in app components. They presented ComDroid to detect app communication vulnerabilities. There also exist works focusing on detecting implemention flaws of the Android framework. Aafer *et. al.* [13] studied the threat of hanging attribute references; Kratos [29] found framework vulnerabilities from the perspective of inconsistent security enforcement. Unfortunately, none of the aforementioned works explore traditional Linux IPCs on Android, *e.g.,* Unix domain sockets, as exploitable interfaces.

Static analysis of Android apps. We use static analysis to detect the misuse of Unix domain sockets in apps. Techniques that serve this purpose have been extensively studied [14, 31, 21, 22]. Particularly, FlowDroid [14] has been widely used for doing taint analysis on Android apps. However, it does not handle inter-component communications (ICC) well. Amandroid [31] is a data flow analysis framework that provides better ICC support, and we build our tool on top of it.

Security risks in customizations. Customizations to the Android framework has been known to introduce new vulnerabilities not present in the AOSP [32]. Wu *et. al.* discovered that over 85% of all preinstalled apps in stock images have more privileges than they need. Of those 85% of apps, almost all vulnerabilities are a direct result of vendor customization. They discovered that many of these firmware and pre-installed apps are susceptible to a litany of vulnerabilities that range from injected malware, to pre-installed malware, and signing vulnerabilities. These all point to a systemic problem introduced by customization of the Android framework. ADDICTED [34] is a tool for automatically detecting flaws exposed by customized driver implementations. On a customized phone, it performs dynamic analysis to correlate the operations on a security-sensitive device to its related Linux files. Our work reveals and studies a new customization domain — privileged system daemons — which can be exploited to perform dangerous operations.

8. CONCLUSION

In this paper, we conducted the first systematic study in understanding the usage of Unix domain sockets by both apps and system daemons as an IPC mechanism on Android, especially for cross-layer communications between the Java and the native layers. We presented SInspector, a tool for discovering potential security vulnerabilities through the process of identifying socket addresses, detecting authentication checks, and performing data flow analysis on na-

tive code. We analyzed 14,644 Android apps and 60 system daemons, finding that some apps, as well as certain system daemons, suffer from serious vulnerabilities, including root privilege escalation, arbitrary file access, and factory resetting. Based on our study, we proposed countermeasures to prevent these attacks from occurring.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on our work. This research was supported in part by the National Science Foundation under grants CNS-1318306 and CNS-1526455, as well as by the Office of Naval Research under grant N00014-14-1-0440.

9. REFERENCES

[1] Advanced audio distribution profile (a2dp). `https://developer.bluetooth.org/TechnologyOverview/Pages/A2DP.aspx`.

[2] An Analysis of Android App Permissions. `http://www.pewinternet.org/2015/11/10/an-analysis-of-android-app-permissions/`.

[3] Android Security Overview. `https://source.android.com/security/`.

[4] Android Security Tips: Using Interprocess Communication. `http://developer.android.com/training/articles/security-tips.html#IPC`.

[5] ApkPure website. `https://apkpure.com/`.

[6] Es app group. `http://www.estrongs.com/`.

[7] How to create a android native service and use binder to communicate with it? http://stackoverflow.com/questions/14215462/how-to-create-a-android-native-service-and-use-binder-to-communicate-with-it.

[8] Jeb decompiler by pnf software. `https://www.pnfsoftware.com/`.

[9] ProGuard. `http://proguard.sourceforge.net/`.

[10] 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.

[11] Security — Platform Security Architecture. `https://source.android.com/security/index.html#android-platform-security-architecture`.

[12] Xposed development tutorial. `https://github.com/rovo89/XposedBridge/wiki/Development-tutorial`.

[13] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proc. of ACM CCS*, 2015.

[14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of ACM PLDI*, 2014.

[15] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Proc. of ISOC NDSS*, 2012.

[16] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. of ACM CCS*, 2007.

[17] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proc. of ACM MobiSys*, 2011.

[18] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proc. of USENIX Security*, 2007.

[19] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Information Security*, pages 346–360. Springer, 2010.

[20] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[21] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Proc. of IEEE S&P*, 2016.

[22] C. Gibler, J. Crussell, J. Erickson, and H. Chen. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*. Springer, 2012.

[23] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proc. of ISOC NDSS*, 2012.

[24] N. Hardy. The Confused Deputy:(or why capabilities might have been invented). *ACM SIGOPS*, 1988.

[25] C.-C. Lin, H. Li, X.-y. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *Proc. of ISOC NDSS*, 2014.

[26] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proc. of ISOC NDSS*, 2008.

[27] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proc. of ACM CCS*, 2012.

[28] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2015.

[29] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proc. of ISOC NDSS*, 2016.

[30] T. Vennon. Android malware. A study of known and potential malware threats. *SMobile Global Threat Centre*, 2010.

[31] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proc. of ACM CCS*, 2014.

[32] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proc. of ACM CCS*, 2013.

[33] H. Zhang, D. She, and Z. Qian. Android root and its providers: A double-edged sword. In *Proc. of ACM CCS*, 2015.

[34] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proc. of IEEE S&P*, 2014.