Exploiting Android System Services Through Bypassing Service Helpers

Yacong Gu[1], Yao Cheng[3], Lingyun Ying[1,2(✉)], Yemian Lu[1], Qi Li[4], and Purui Su[1,2]

[1] Institute of Software, Chinese Academy of Sciences, Beijing, China
{guyacong,luyemian,yly,supurui}@tca.iscas.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Singapore Management University, Singapore, Singapore
ycheng@smu.edu.sg
[4] Graduate School at Shenzhen, Tsinghua University, Shenzhen, China
qi.li@sz.tsinghua.edu.cn

Abstract. Android allows applications to communicate with *system service* via *system service helper* so that applications can use various functions wrapped in the system services. Meanwhile, system services leverage the service helpers to enforce security mechanisms, e.g. input parameter validation, to protect themselves against attacks. However, service helpers can be easily bypassed, which poses severe security and privacy threats to system services, e.g., privilege escalation, function execution without users' interactions, system service crash, and DoS attacks. In this paper, we perform the first systematic study on such vulnerabilities and investigate their impacts. We develop a tool to analyze all system services in the newly released Android system. Among the 104 system services and over 3,400 system service methods in the system, we discover 22 vulnerable service interfaces that can be exploited to launch real-world attacks. Furthermore, we implement and construct attacks to demonstrate the impacts of these vulnerabilities. In particular, by utilizing these vulnerabilities, these attacks result in implicit user fingerprint authentication in background, NFC data retrieval in background, Bluetooth service crash, and Android system crash.

Keywords: Android · System services · Service helpers · Vulnerabilities

1 Introduction

One of the most salient features in Android is that it wraps various functions in its *system services*, such as telephony, notification, and clipboard, so that different applications ("apps") can easily access these functions through inter-process communication (IPC). Normally, apps use these system services via *system service helper*. In order to protect system services, service helpers provide various security mechanisms so as to protect the system services, e.g., validating input

R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 44–62, 2017.
DOI: 10.1007/978-3-319-59608-2_3

parameters against service crash, checking callers' status against user authentication in background, identifying and handling duplicated requests against unnecessary resource consumption, and passing callers' identities (IDs) to allow system services to authenticate the callers.

However, we find that these mechanisms can be easily bypassed, which incurs serious security problems. For example, as shown in Fig. 1, `FingerprintManager`, i.e., the service helper of system service `FingerprintService`, automatically obtains a caller's identity (i.e., the app's package name) and passes it to `FingerprintService` so that the service could enforce particular restrictions based on the caller's identity. Unfortunately, a malicious app can bypass the service helper and directly feed fake ID to `FingerprintService`. Therefore, the system service will directly accept the fake ID without any authentication. As we observed, the vulnerabilities of bypassing service helpers can incur privilege escalation, automatic function execution without user interaction, system service crash, and Denial-of-Service (DoS) attacks. Therefore, it is necessary to systematically study such vulnerabilities and their impacts, which has not yet been well studied in the literature.

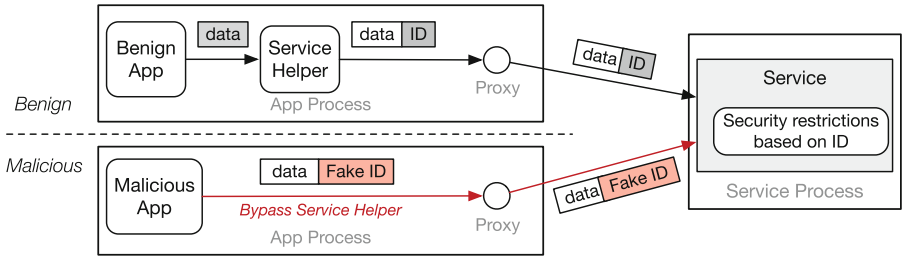

Fig. 1. A benign app interacts with a system service through the corresponding service helper which automatically collects the caller's identity and passes it to the system service. However, a malicious app can bypass the service helper, and directly feed a fake identity to the system service.

In this paper, we perform a systematic study on the above security breaches related to service helper bypass. The root cause of the breaches is that system services assume correct execution of security mechanisms in the corresponding service helpers that actually can be bypassed. In order to find out all vulnerable IPC methods in system services that can be exploited because of service helper bypass, we develop a four-step approach to identify IPC methods that do not enforce security mechanisms corresponding to that in the service helpers. Firstly, we enumerate all system services as well as their IPC methods in the Android source code. Secondly, we identify the corresponding service helper classes for each of the services. Here, we need to consider internal and hidden APIs which can be invoked by third-party apps through Java reflection. By scrutinizing the source code, we extend Android SDK and define service helpers as the classes in the extended SDK that can access system services via IPC methods.

The extended SDK includes all APIs accessible to third-party apps. Thirdly, after obtaining system services and their helpers, we identify the presence of security mechanisms for each method in two categories of classes, i.e., system services and the corresponding service helpers, by applying static analysis. Since service helpers run in the same process with the calling apps, while services do not, we cannot use the same method to identify the mechanisms in these two classes. To address this issue, we extract and compare code features of security mechanisms in the system services and the corresponding service helpers so as to obtain the difference in the security mechanisms. Finally, vulnerabilities are detected if system services do not enforce security mechanisms that are enabled in the corresponding service helpers.

We study the vulnerabilities in the Android 6.0.1. We find 22 vulnerabilities in system services resulted from bypassing service helpers that can be exploited by third-party apps to launch real-world attacks. We have submitted all vulnerabilities to Android Security Team and got confirmed by multiple Android Bug IDs. To demonstrate the impacts of the vulnerabilities, we exploit several representative vulnerabilities by constructing real-world attacks that lead to user fingerprint authentication in background, NFC data retrieval in background, Bluetooth service crash, and Android system crash.

In summary, the contributions of this paper are three-fold.

- To the best of our knowledge, this paper performs the first systematic study on security problems incurred by bypassing system service helpers. We find that bypassing service helpers can lead to the abuse of system services.
- We propose a method to identify the vulnerabilities in system services that are caused by bypassing service helpers and could be exploited to launch real-work attacks.
- We identify 22 vulnerabilities in total in Android 6.0.1, all of which are confirmed by Android Security Team. Moreover, we construct several malicious apps to exploit the vulnerabilities to illustrate the impacts of the vulnerabilities.

2 Background

In this section, we briefly review system services and service helpers, and the security mechanisms enabled in service helpers.

2.1 System Service vs. Service Helper

System services encapsulate essential functions in Android and compose significant parts of Android Open Source Project (AOSP) [1]. Two representatives of the functions in the form of system service are Near Field Communication (NFC) service and notification service. Usually, system services run as system processes and are registered in the *service manager* which serves as a service information center. Apps that intend to use a system service can query available services in

the service manager, and then obtain the service proxy object through which functions in the system service can be used.

Service helpers enabled in Android SDK provide interfaces of functions in the system services for apps so that apps can easily use the functions, e.g., service helpers can automatically feed in parameters for functions in the system services. Most of system services that can be accessed by third-party apps are with one or more service helpers. In order to protect system services, the corresponding service helpers also use some security mechanisms to validate the requests from apps before issuing an IPC call to the system services.

A data flow of accessing system service is shown in Fig. 1. Since service helpers run in the same process with the calling app, an app directly calls the service helper within its own process. The service helper acquires the corresponding service proxy from the service manager, and then sends the request to the service proxy that is responsible for communicating with the target service via IPC calls so as to execute the system functions. The detailed communication procedures between services and the corresponding service proxies are defined by Android Interface Definition Language (AIDL) [2].

2.2 Security Mechanisms in Service Helpers

In order to ensure security, reliability, and efficiency of system services, service helpers include the following mechanisms. First of all, they enable the fail-fast principle [3] to ensure system reliability. As providing direct interfaces to apps, service helpers should detect failures as early as possible. We find that service helpers validate parameters and the caller's status to prevent service failures incurred by wrong parameters and status. Secondly, service helpers automatically collect data required by the system services, e.g., passing caller's identity, which reduces the risk incurred by passing invalid parameters to services. Meanwhile, they decrease the number of parameters that apps need to feed in. Lastly, service helpers help the system services to deal with duplicated requests that waste the service resources. Now we use four examples to illustrate the typical protection enabled by service helpers to protect the system services.

Validating Input Parameters. Parameter validation is one of the most important security mechanisms. For example, `BluetoothHealth`, the helper of the system service `BluetoothHealthService`, checks in `BluetoothHealth.registerAppConfiguration (String name, int dataType, ...)` to verify whether the first string parameter is non-null. It will send the remote request to `HealthService` via its service proxy only if the parameter is not empty.

Validating Callers' Status. Some Android system services may be allowed to be used only when the callers are currently active in the foreground. Service helpers provide such assistance to verify caller's status. One example is `NfcAdapter`, which is the helper of system service `NfcService`, verifies a caller's status in `NfcAdapter.enableForegroundDispatch()`. If the caller is not currently active in the foreground, the registration request for using NFC listeners will be rejected.

Passing Callers' Identities. System services need to verify various identity information of the callers (see Sect. 3.3.1), such as caller's `uid`, which can be achieved by interacting with `Binder` [4]. For other information, such as package name, system services rely on their helpers to collect. For instance, in the notification service, the first parameter of `NotificationManagerService.enqueueToast (String pkg, ITransientNotification callback, int duration)` is collected by one of its helpers `Toast`. Note that, such information delivery is transparent to apps because it is automatically performed by service helpers.

Constraining Duplicated Requests. There may exist multiple IPC requests from an app to a system service during the app's lifetime. IPC requests will consume the limited system resources, such as memories, CPU time, and file descriptors. Therefore, if a system service accepts all duplicated requests, the limited resources may be exhausted. In order to prevent resource being exhausted by duplicated requests, service helpers handle duplicated calls locally and mitigate the impacts. Service helpers use two strategies to constrain duplicated requests. For resource requests, service helpers restrict the number of calls that an app can issue. If the number of duplicated calls exceeds a threshold, which can be treated as abnormal or unnecessary, the following calls will be ignored locally. While for requests of registering listeners, service helpers initiate an IPC call to the remote system service only under receipt of the first request from the app. For instance, an app can get a notification when the clipboard changes by registering a listener to the system service `ClipboardService`. `ClipboardManager`, which is the service helper of `ClipboardService`, only registers the service once to get such listener after receiving the first request. After that, `ClipboardManager` maintains a local listener queue. Any duplicated requests of registering listeners afterwards will be only added to this queue locally. When the clipboard changes, `ClipboardService` notifies its helper `ClipboardManager` of the change. Then, `ClipboardManager` notifies all the listeners in its local queue. Thereby, system services only allocate resources for one request from the app, but still can notify the app when there is any update.

Unfortunately, these mechanisms can be easily bypassed if a malicious app directly invokes methods in system services instead of that in the service helpers. In this paper, we will focus on the vulnerabilities incurred by bypassing service helpers and the consequent impacts on the system services.

3 Identifying Vulnerabilities

In this section, we present an approach to systemically studying the vulnerabilities caused by bypassing service helpers.

3.1 Overview

The main idea of our approach is to find out whether the security mechanisms enforced by system services are consistent with that in their service helpers.

Fig. 2. Overview of our approach.

If they are not consistent, we regard it as a potential vulnerability that may be exploited through bypassing the service helper. For each potential vulnerability, we need to manually confirm it since not all potential vulnerabilities can be exploited. Following this, our study based on the source code of Android composes of four steps as shown in Fig. 2.

Firstly, we identify all system services as well as their IPC methods that can be accessed by third-party apps. We need to consider the services both in framework layer and native layer. We leverage service manager, in which all services are registered, as choke points to obtain all system services. We obtain IPC methods for most services that can be extracted from the AIDL files. Note that, a small portion of services (i.e., five system services) do not have their public IPC methods in the AIDL files. We manually extract their IPC methods.

Secondly, for each service, we need to find out its service helper class(es). We extend Android SDK to define service helpers as the classes that request services via IPC methods and can be accessed by third-party apps. We associate system services with their service helpers in the method level. If a service helper method includes an IPC call to invoke a remote service method, we treat these two methods as a pair. Here, we need to take all APIs into consideration, including the internal and hidden ones that can be invoked by third-party apps using Java reflection.

Thirdly, we examine the presence of security mechanisms in both system services and service helpers. We extract the code features of different security mechanisms. In this step, we need to detect the presence of security mechanisms in system services and their helper classes separately due to their differences.

Finally, potential vulnerabilities are detected by comparing whether the security mechanisms in the method pairs are consistent. If they do not match, it means that the service includes a potential risk of being exploited by bypassing service helpers and the inside security mechanisms. We manually confirm the vulnerabilities by launching real attacks.

There are two major challenges to identify all vulnerabilities. Firstly, it is not easy to find out all service helper classes. The internal and hidden APIs [5] that can be invoked by third-party apps through Java reflection are not included in the official Android SDK. If we intend to find out all service helper classes that third-party apps can access, we need to consider all APIs. Secondly, it is difficult to identify security mechanisms in system service methods and the corresponding service helper methods. The security mechanisms in service helpers and the system services are implemented with different methods, in particular, service helpers run in the same processes with the caller app, whereas system services run in a separate system process. Therefore, we cannot directly compare the

source code of security mechanisms in system services and their corresponding service helpers.

3.2 Enumerating Service Helper Classes

We show how to find service helper classes for various system services. Since the standard Android SDK does not include the internal and hidden APIs that can be invoked by third-party apps through Java reflection, we cannot directly enumerate all the APIs. To address this, we extend Android SDK to define service helpers as the classes that can use the corresponding system services via IPC methods. Internal APIs are located in `com.android.internal` package which is available in the `framework.jar` file on real Android device, while hidden APIs are located in `android.jar` file with `@hide` javadoc attribute. We merge the `android.jar` file and the `framework.jar` file to generate the *extended* SDK which includes all APIs that can be directly accessed by third-party apps.

We use Soot [6] to automatically analyze all classes in the extended SDK. One class is treated as a service helper as long as it invokes an IPC method of a system service by using one or more methods. If an identified class is a nested class, an inner class, a local class or an anonymous class, the top level enclosing class is considered as the service helper class. We further associate the service helper method with the service IPC method, and these methods compose a method pair that will be used to analyze security mechanisms in a later subsection.

3.3 Detecting Security Mechanisms

We identify the presence of security mechanisms in these methods by constructing a call graph and detecting their code features in the graph. We construct call graphs to express the relationships inside system services and service helpers and that between them. The call graphs are constructed by using Soot on the method level according to system services and service helpers. Moreover, we use PScout [7] to parse indirect dependency, e.g., *Message Handler* invokes different methods to handle messages according to the message content, so as to construct complete call graphs. Note that, in order to allow our approach to work with the latest Android 6.0, we also adopt the new compiling strategy in Android 6.0. Since Android 6.0, AOSP adopts a new Java Android Compiler Kit (Jack) toolchain [8] to generate `.jack` and `.dex` files as build targets. Since PScout uses `.jar` files as default input, we need to convert `.dex` files to `.jar` files by using `dex2jar` tool [9].

3.3.1 Identify Security Mechanisms in Service Helpers

We use different methods to identify the presence of the four types of security mechanisms in service helpers, separately.

Identifying Parameter Validation. We adopt `def-use` analysis [10] to identify the parameter validation mechanisms. `Def-use` analysis links each variable definition with that is referred, which can be used to identify if the variable, i.e.,

an input parameter, is referred in a validation process. Firstly, we check whether input parameters of a method are used in boolean expressions or whether they are used in other methods that return boolean values. Secondly, if the parameters are indeed related to boolean values, we further verify whether the boolean values are used in conditional statements, which contain statements with early returns or thrown exceptions. If these two conditions are met, the method includes input parameter validation.

Identifying Caller Status Validation. For the caller status validation, it is similar to the input parameter validation from the perspective of code features. We can identify caller status validation by analyzing APIs that return callers' status and verifying if these APIs are used in conditional statements.

Identifying the Process of Passing Caller's Identity. Apps provide different types of identities. As we observed, there are seven types of identity information in Android, i.e., package name, `uid` (i.e., linux user identifier), `pid` (i.e., linux process identifier), `gid` (i.e., linux group identifier), `tid` (i.e., linux thread identifier), `ppid` (i.e., linux parent process identifier), and `UserHandle` (i.e., representing a user in Android that supports multiple users). Each of them can be obtained by calling relevant methods that are summarized in Table 1. If there is any method included in a service helper method before an IPC method of the target service in the call graph, there is a high possibility that this service helper will pass a caller's identity to its service.

Identifying the Constraint of Duplicated Requests. In order to prevent resource consumption incurred by duplicated requests from apps, service helpers

Table 1. Methods used by service helpers to obtain caller's identity.

Identity type	Method
Package name	Context.getPackageName()
	Context.getBasePackageName()
	Context.getOpPackageName()
UID	Process.myUid()
	Process.getUidForPid()
	Context.getUserId()
PID	Process.myPid()
	Process.getPids()
	Process.getPidsForCommands()
GID	Process.getGidForName()
	Process.getProcessGroup()
PPID	Process.myPpid()
	Process.getParentPid()
TID	Process.myTid()
UserHandle	Process.myUserHandle()

adopt two ways to constrain the number of duplicated requests delivered to the system services according to the type of requested resources (see Sect. 2.2). The first approach is to restrict the number of requests that an app can issue to request resources. If the total number of requests exceeds a threshold, the following requests would be ignored. To identify the existence of this approach, we search all methods in service helpers to locate the conditional statements where the conditions are with constant integer expressions. If such conditional statement is located after the entry of the corresponding service helper and before the IPC calls to the service in the call graph, there is a high probability that the statement is used to check duplicated requests, which is similar to input parameter validation. Therefore, we can use a similar method to further confirm the detected mechanisms.

The second approach is to constrict the number of duplicated requests to register listeners. Usually, a service helper method accepts a listener as its parameter, and saves the listener to a local list. For example, the service helper method, `EthernetManager.addListener(Listener listener)` saves the parameter listener to `ArrayList <Listener> mListeners`. If it is the first registration request, the helper will register in the remote service via IPC. Otherwise, the service helper method only adds the request's listener to the local list. When the service helper receives the update from the service, it dispatches the update to all the listener maintained in that list. We capture the mechanism by identifying the code maintaining the listener lists.

3.3.2 Identifying Security Mechanisms in System Services

The idea of identifying security mechanisms in service helpers can be used to identify the security mechanisms enforced in system services. However, we cannot directly adopt the methods in Sect. 3.3.1 to system services because of the difference between systems services and service helpers.

Firstly, service helpers run in the same process with the caller while services do not. Service helpers can directly obtain the caller's identity via methods in Table 1. However, system services need different APIs to obtain the information about the caller and check the caller's properties, since they run in system processes which are separate from the caller processes. For instance, system services use `Binder.getCallingUid()` and `Binder.getCallingPid()` to obtain the callers' identities instead of the methods listed in Table 1. Secondly, system services need to validate app identities and verify whether the calling app is privileged to perform sensitive operations, which is not required in the service helpers. Fortunately, we find that system services heavily rely on the functions provided by `AppOpsService` to perform validation. For example, `AppOpsService.checkPackage(int uid, String packageName)` checks whether the input package name actually belongs to the given uid, and `AppOpsService.checkOperation(int code, int uid, String packageName)` checks whether the `uid` has the privilege to perform the sensitive operation indicated by the `code`. We can use these key methods to

Table 2. Methods used by services to obtain and check identity.

Function	Method
Get caller's UID	Binder.getCallingUid()
Get caller's PID	Binder.getCallingPid()
Get caller's UserHandle	Binder.getCallingUserHandle()
Check package name	AppOpsService.checkPackage()
Check operation	AppOpsService.checkOperation()

identify the process of verifying identities in system services. These key methods in system services are listed in Table 2.

3.3.3 Detect Possible Vulnerabilities

The final step is to examine whether the security mechanisms are consistent in the method pairs we identified in Sect. 3.2, i.e., the service method and the corresponding service helper method. The examination is straightforward and can be automated in most cases. Taking parameter validation as an example, we separately form the parameters validated in each party of the method pair into two sets. The parameters validated in the service method are denoted as set S, and those in the service helper method are denoted as set H. If S is *not* the superset of H, which means the helper checks more parameters than the service, the parameters p, which belongs to H but does not belong to S, may be abused with illegal values. For the processes of dealing with duplicated calls which are few in numbers (9 methods of 7 system helpers), we manually verify the enforcement consistency within the pairs.

4 Vulnerability Results

We develop a tool based on the methodology in Sect. 3, and apply it to analyze the latest AOSP 6.0.1. This section firstly summarizes our experimental findings, including the vulnerabilities related to the four types of security mechanisms in service helpers. Then, we construct real-world attacks exploiting representative vulnerabilities.

4.1 Vulnerability Summary

In the extended SDK containing 8130 classes, we find out 158 service helper classes. Among these service helpers, there are 86 cases where the helper passes caller's identity to the service. Also, these helpers classes include 227 methods that validate input parameters, six methods that verify caller's status, and nine methods that handle duplicated requests. Among these service helpers, as shown in Table 3, we capture 22 vulnerabilities, which can lead to privilege escalation, bypass of user interactions, service crash, or Android system soft reboot. All these

Table 3. Summary of vulnerabilities resulted from bypassing service helpers.

Service helper	Vulnerable service method	Security implication	Type
Toast	enqueueToast	Soft reboot	Fake identity
NotificationManager	setNotificationPolicy	Privilege escalation	Fake Identity
	getNotificationPolicy		
	setInterruptionFilter		
FingerprintManager	authenticate	Privilege escalation	Fake identity
	cancelAuthentication		
	getEnrolledFingerprints		
	hasEnrolledFingerprints		
	isHardwareDetected		
MediaBrowser	addSubscription	DoS	Illegal parameter
	removeSubscription		
BluetoothHealth	registerAppConfiguration	DoS	Illegal parameter
NfcAdapter	enableForegroundDispatch	Bypass of user interaction requirements	Fake status
ClipboardManager	addPrimaryClipChangedListener	Soft reboot	IPC flooding
AccessibilityManager	addClient	Soft reboot	IPC flooding
LauncherApps	addOnAppsChangedListener	Soft reboot	IPC flooding
TvInputManager	registerCallback	Soft reboot	IPC flooding
EthernetManager	addListener	Soft reboot	IPC flooding
WifiManager	WifiLock.acquire	Soft reboot	IPC flooding
	MulticastLock.acquire		
LocationManager	addGpsMeasurementsListener	Soft reboot	IPC flooding
	addGpsNavigationMessageListener		

vulnerabilities have been confirmed by Android Security Team and assigned with different Android Bug IDs. We describe the vulnerabilities in the following.

Vulnerabilities Caused by Passing Fake Identity. We find 19 inconsistent method pairs that are identified in 86 cases where service helpers pass the callers' identities to the services. That is, 19 service methods receive callers' identities from the corresponding service helpers but fail to verify the authenticity of the received identities as the service helpers do. We manually verify whether all of them can be exploited. Our verification shows that nine out of the 19 inconsistent method pairs can be used to launch real-world attacks. The rest 10 methods are not vulnerable to the fake identity attacks. Among them, five methods are protected by high level permissions (i.e., `signature` and `signatureOrSystem` levels) and cannot be granted to third-party apps, such as `StatusBarManager.setIcon()`, and the other five methods do not incur security issues, such as `BackupManager.dataChanged()`.

One vulnerability is abuse of `enqueueToast()` in notification service, which can lead to system reboot. Malicious apps will be regarded as one of the system apps by passing a fake package name "android" and can exhaust the system resources. The other eight vulnerabilities are in the `notification` service and

the `fingerprint` service, which will result in the privilege escalation. A real-world attack to `fingerprint` service is illustrated in Sect. 4.2.1.

Vulnerabilities Caused by Passing Illegal Parameter. We find out 227 service helper methods validating their input parameters. Among these service helper methods and their corresponding service IPC methods, 51 method pairs are inconsistent in validating the input parameters. These methods may be exploited. After manual verification, three methods are identified to be vulnerable, i.e., they can be exploited to crash their services.

The reason most of inconsistent method pairs are secure is that Android automatically adds handle code for some common exceptions when generating Java classes from AIDL. The six most common exceptions are well handled in the system services defined by AIDL, including `BadParcelableException`, `IllegalArgumentException`, `IllegalStateException`, `NullPointerException`, `SecurityException`, and `NetworkOnMainThreadException` [11]. For other exceptions, it re-throws them as `RuntimeException`s. The exceptions thrown by system service IPC methods are caught by the Binder framework. The Binder framework then passes the exception to the IPC caller through `Parcel.writeException()` so that the caller could handle these exceptions in its own process. Therefore, if the exception triggered by illegal parameters occurs inside the service IPC methods, it would be handled well and not crash the service. However, if the parameter is used outside the service IPC methods, such as being used in asynchronous handler or stored for later access, it may lead to security issues due to the failure of handling exceptions. These three vulnerabilities in `MediaBrowserService` and `HealthService` are due to such failure in handling exceptions. In Sect. 4.2.3, we construct a real-world attack to illustrate the process of crashing Bluetooth service by passing illegal parameters.

Vulnerabilities Caused by Invoking IPC with Fake Status. We find that six service helper methods check the caller's status. One of them lacks the validation of the caller's status in its corresponding service method, i.e., `NfcService`. It bypasses user interaction to access function without user initiation or user permission, we will show such a case in Sect. 4.2.2 that an app can retrieve NFC data in background.

Vulnerabilities Caused by IPC Flooding. We identify nine methods in service helpers that handle duplicated requests which can be bypassed. These helper methods firstly check whether the current request is a duplicated one. These methods handle duplicated requests either by processing the requests locally (but not delivering them to the services) or restricting the number of the requests that can be delivered to the services. However, as we point out, these methods can be easily bypassed by directly using the methods in the corresponding services. A malicious app can abusively invoke corresponding service via IPC without any restriction. A large number of IPC calls would lead to Android resource exhaustion, which can further cause the system reboot. An attack leading to Android soft reboot will be described in Sect. 4.2.1.

4.2 Real-World Attacks

In this section, we demonstrate the impacts of several representative vulnerabilities by constructing real-world attacks to exploit these vulnerabilities.

4.2.1 User Fingerprint Authentication in Background

The `FingerprintService` provides functions related to user fingerprint authentication. We discover that there are five vulnerable methods in `FingerprintService` which all result in the privilege escalation by passing fake identify. The functions include fingerprint authentication (`authenticate()` and `cancelAuthentication()`), accessing the information about the enrolled fingerprints of a particular user (`getEnrolledFingerprints()` and `hasEnrolledFingerprints`), and determining if the fingerprint sensor is present and functional on the current device (`isHardwareDetected()`).

Take `FingerprintService.authenticate()` as an example, the helper class of `FingerprintService`, i.e., `FingerprintManager`, is responsible for automatically collecting and passing the caller's package name to `FingerprintService`. `FingerprintService.authenticate()` is used to authenticate a given fingerprint. As shown in Listing 1, in `authenticate()`, it verifies whether the caller is allowed to use fingerprint based on received package name (Line 2). In `canUseFingerprint()`, it evaluates whether the caller is the current user or profile (Line 12), whether `App Ops` allows the operation (Line 14), and whether the caller is currently in the foreground (Line 16). Note that if the caller's package name is of `KeyguardService`, the caller is always allowed to use the fingerprint, and above restrictions could be bypassed (Line 10). Unfortunately, `authenticate()` never verify the authenticity of received package name. A malicious app can bypass the service helper to directly feed the `KeyguardService`'s package name to the service method `FingerprintService.authenticate()`. In this case, a malicious app can circumvent these three restrictions in `canUseFingerprint()`. This vulnerability is confirmed with Bug ID AndroidID-29324069.

```
1  public void authenticate(/* other parameters */, String pkgName) {
2      if (!canUseFingerprint(pkgName, true)) {
3          Slog.v(TAG, "authenticate(): reject " + pkgName);
4          return;
5      }
6      ....
7  }
8  boolean canUseFingerprint(String pkgName, boolean foregroundOnly) {
9      checkPermission(USE_FINGERPRINT);
10     if (pkgName.equals(mKeyguardPackage))
11         return true; // Keyguard is always allowed
12     if (!isCurrentUserOrProfile(UserHandle.getCallingUserId()))
13         return false;
14     if (mAppOps.noteOp(OP_USE_FINGERPRINT, uid, pkgName) != MODE_ALLOWED)
15         return false;
16     if (foregroundOnly && !isForegroundActivity(uid, pid))
17         return false;
18     return true;
19 }
```

Listing 1: Code snippet in `FingerprintService.java`
(All code we present in this section has been simplified for brevity).

4.2.2 NFC Data Retrieval in Background

The `NfcService` provides NFC operations such as reading data from a close NFC tag. The service helper of `NfcService` checks the app's status by its method, i.e., `NfcAdapter.enableForegroundDispatch()`, so that only the app currently running in the foreground could register NFC listeners (see Listing 2). It is a rational design that the activity currently in the foreground should be the preferential destination for new coming NFC events. However, a malicious app could directly call service proxy method `INfcAdapter.enableForegroundDispatch()` to register an NFC listener. This is different from passing fake identities. There is no need to feed a fake status as the status is not passed to the service side. The NFC service does not verify whether the app is indeed in the foreground. In this case, the malicious app in the background can successfully register NFC listeners. When an NFC tag approaches the device, the malicious app in the background can read data on the NFC tag which may lead to user privacy leakage. We have reported this vulnerability to Android Security Team. It is tracked with AndroidID-28300969 with moderate severity level.

```
1  /**
2   This method must be called from the main thread,
3   and only when the activity is in the foreground (resumed).
4  */
5  void enableForegroundDispatch(Activity aty, /* other paramters */) {
6      if (!aty.isResumed()) {
7          throw new IllegalStateException("Foreground dispatch can only be
               ↪  enabled " + "when your activity is resumed");
8      }
9      sService.setForegroundDispatch(intent, filters, parcel);
10 }
```

Listing 2: Code in `NfcAdapter.java`, the helper class of NFC service.

4.2.3 Bluetooth Service Crash

The `HealthService` service, which provides health-related Bluetooth service, contains a vulnerability that can be exploited by passing illegal parameters. The method pairs, i.e., the service helper method `BluetoothHealth.registerAppConfiguraton(String name, int dataType...)` and the corresponding service method `HealthService.BluetoothHealthBinder.registerAppConfiguration(String name, int dataType...)`, do not use the same method to validate parameters. The helper method checks the "name" parameter to make sure it is not null, whereas the service method does not. The service does not use the value of "name" parameter immediately in the IPC method. Instead, it stores the value and uses it in `BluetoothHealthAppConfiguration.equals()`. In `BluetoothHealthAppConfiguration.equals()`, it assumes `config.getName()`, that returns the value of "name", can never be null as shown in Listing 3. When a malicious app bypasses the service helper and registers a config with null-value in "name" parameter, there would be a `NullPointerExcetion` in `BluetoothHealthAppConfiguration.equals()`. Unfortunately, this method fails to handle the exception, and hence

the `HealthService` crashes. This vulnerability is acknowledged by Android Security Team, and tracked as AndroidID-28271086.

```
public boolean equals(Object o) {
    if (o instanceof BluetoothHealthAppConfiguration) {
        BluetoothHealthAppConfiguration config = o;
        // config.getName() can never be NULL
        return mName.equals(config.getName()) && ... ;
    }
    return false;
}
```

Listing 3: Code snippet in `BluetoothHealthAppConfiguration.java`. The possible reason is that Google engineers assume that `mName` should never be NULL since it is validated in its corresponding helper class.

4.2.4 Android System Crash

This is an attack related to restriction on duplicated requests. The helper class associated with Wi-Fi service is `WifiManager`. An app can acquire Wi-Fi lock to prevent Wi-Fi to go in stand-by. This is done by calling `WifiManager.WifiLock.acquire()`. The source code is shown in Listing . This method examines whether the current request exceeds the maximum lock number that an app can acquire. If it detects that the current request has exceeded the threshold, it would release the lock immediately. We can see from the comments in AOSP that the restriction here is to "prevent apps from creating a ridiculous number of locks and crashing the system by overflowing the global ref table." However, a malicious app can easily bypass such restriction in the service helper by directly issuing requests to Wi-Fi service via the service proxy, i.e., `IWifiManager.acquireWifiLock()`. A large number of IPC calls would overflow the reference table, and lead to the crash of Wi-Fi service and then the reboot of Android. This vulnerability is tracked as AndroidID-27596394.

```
/* Maximum number of active locks we allow.
 * This limit was added to prevent apps from creating a ridiculous number of
 ↪   locks and crashing the system by overflowing the global ref table.
 */
private static final int MAX_ACTIVE_LOCKS = 50;
.....
public void acquire() {
    mService.acquireMulticastLock(mBinder, mTag);
    if (mActiveLockCount >= MAX_ACTIVE_LOCKS) {
        mService.releaseWifiLock(mBinder)
        throw new Exception("Exceeded maximum number of wifi locks");
    }
    ...
}
```

Listing 4: Code snippet in `WifiManager.java`.

5 Discussion

5.1 Lessons Learned

Service helpers play an important role in assisting both app developers (for easy app development) and services (for security verification). Unfortunately, we show that the use of the service helpers could be manipulated. In a manipulated process, service helpers can be bypassed. It means all the security mechanisms would be in vain. We have identified that there are indeed a large number of such vulnerabilities (Sect. 4). These vulnerabilities can lead to privilege escalation, bypass of user interaction requirements, service crash, and Android system soft reboot.

All vulnerabilities discussed in our paper are incurred by that security mechanisms in service helpers are bypassed. Since Android cannot guarantee a control flow to the service is initiated by a service helper, to completely prevent the attacks, an intuitive solution is to let services enforce the same security mechanisms as that in the corresponding service helpers, i.e., verifying callers' identities, verifying callers' status, validating input parameters, or constraining duplicated requests.

5.2 Limitations

False Negatives. Even though we have detected a series of vulnerabilities caused by the bypass of service helpers, we have to admit that there may be more such vulnerabilities to be uncovered. There are two factors leading to false negatives. One factor is that we do not consider the sequence of check and use. In our approach, we examine whether the service enforces the same security mechanisms as the service helper does. We assume that as long as there are the same enforcements in services, the adversary cannot abuse the service even though (s)he can bypass its service helpers. However, some defective services may perform sensitive operations before the security checking, e.g., using the parameter before the validation. In this case, the early use may lead to potential risks even with the presence of the same verifications in services. Another factor is that some services invoke native code using JNI, which is a small portion of the service code [5]. We do not study on these native code in our analysis, which also results in false negatives. We leave the analysis on services with JNI native code as future works.

Manual Work. Our approach is mostly automated, but still involves some manual works. The manual works are used in three processes. The first one is to identify the IPC methods that are not defined by AIDL. Fortunately, there are only five such services implementing their own IPC methods without AIDL. The second manual work is to examine whether the process of dealing with duplicated request are consistent in method pairs. There are nine pairs in total that need to be manually verified. The third one is to verify whether the experimental results can indeed be exploited to launch real-world attacks. It is also necessary to investigate the impacts of the vulnerabilities by verifying if they can be exploited.

6 Related Work

Vulnerabilities in Android have been extensively studied, including private data leakage [12–16], privilege escalation [17–21] and component hijacking [22–25]. In this section, we only summarize and compare with the existing studies closely related to ours.

Android System Service Security. There have been only a few works [5, 26,27] on the security of system services, despite the significant part they take in Android framework and the important role they play in Android. Huang et al. [26] discover a design flaw in the concurrency control of Android system services. They notice that Android system services often use the lock mechanism to protect critical sections or synchronized methods. If an application takes a lock for a long time, other services sharing the same lock would freeze, and then the watchdog thread would force Android to reboot. Shao et al. [5] find out that there are multiple execution paths leading to the same system service function but with inconsistent privilege requirement. Malicious apps can escalate their privileges or even perform DoS attacks by redirecting their requests to the paths with less enforced permissions. Different from the study only on the service side, our work focuses on investigating the impacts of bypassing service helpers by studying the security mechanisms in both services and service helpers. Another related work [27] examines the input validation in system services using fuzzing. They have identified several DoS attacks due to the lack of proper input validation in system services. Parts of our work is related to the input validation of system services. The identity collected by the service helper and the parameters prepared by the developer are passed to the service as its input. Our relevant findings in Sects. 4.2.1 and 4.2.3 reveal more vulnerabilities in the parameter validation which are missed in their detection. These studies are unable to discover the vulnerabilities since they can be exploited by constructing special parameters. For example, the `FingerprintService` service can be exploited if the input parameter is set to be the package name of `KeyguardService` (see Sect. 4.2.1. However, it is not easy for fuzzing to construct the parameters so as to effectively find this vulnerability.

Static Analysis in Android. Static analysis is one of the most effective ways to analyze the vulnerabilities in both Android systems and apps. Based on static analysis, there have been various researches on malware detection [28–30], library security [31], repackaging detection [32,33], component security [25], system service security [5,26], and permission specification [7]. Static analysis tools [6,34,35] also have been proposed to solve different problems. The two [5,26] closely related to our work both use static analysis. The difference is that our study focuses on the security breaches related to bypassing service helpers. Moreover, since static analysis cannot accurately reflect the precise situations in runtime, the analysis results may not be accurate. In our paper, we verifies the found vulnerabilities by constructing real-world attacks.

7 Conclusion

To our best knowledge, our study is the first systemic study on security problems of bypassing service helper of various Android system services. We point out that system services face the risk of being abused via bypassing the security mechanisms in service helpers. In order to identify such vulnerabilities and demonstrate the impacts of vulnerabilities, we develop a tool to analyze the system services in the latest AOSP. The experimental results reveal 22 vulnerabilities that can be used to launch real-world attacks.

Acknowledgments. This work was partially supported by the National Natural Science Foundation of China under grants 61572278, 61502468, 61502469 and 61572483, the National Key R&D Program of China under grant 2016YFB0800102, and the National Basic Research Program of China (973 Program) under grant 2012CB315804.

References

1. Android open source project. https://android.googlesource.com/
2. Android interface definition language. https://goo.gl/UFrnT3
3. Gray, J.: Why do computers stop and what can be done about it? In: Symposium on Reliability in Distributed Software and Database Systems (1986)
4. Android API reference: Binder. https://goo.gl/w2fXFH
5. Shao, Y., Chen, Q.A., Mao, Z.M., Ott, J., Qian, Z.: Kratos: discovering inconsistent security policy enforcement in the android framework. In: Proceedings of the 23rd NDSS (2016)
6. Soot. https://sable.github.io/soot/
7. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: Proceedings of the 19th CCS (2012)
8. Compling with jack. https://goo.gl/o9RYX8
9. Dex2jar. https://goo.gl/skfQLl
10. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques, and Tools. Addison Wesley, Boston (1986)
11. Android API reference: Parcel.writeexception(). https://goo.gl/7zuXuR
12. Rasthofer, S., Arzt, S., Bodden, E.: A machine-learning approach for classifying and categorizing android sources and sinks. In: Proceedings of the 21st NDSS (2014)
13. Cai, L., Chen, H.: Touchlogger: inferring keystrokes on touch screen from smartphone motion. In: Proceedings of the 6th HotSec (2011)
14. Xu, Z., Bai, K., Zhu, S.: Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In: Proceedings of the Fifth WISEC (2012)
15. Aviv, A.J., Sapp, B., Blaze, M., Smith, J.M.: Practicality of accelerometer side channels on smartphones. In: Proceedings of the 28th ACSAC (2012)
16. Cheng, Y., Ying, L., Jiao, S., Su, P., Feng, D.: Bind your phone number with caution: automated user profiling through address book matching on smartphone. In: Proceedings of the 8th ASIACCS (2013)
17. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., Shastry, B.: Towards taming privilege-escalation attacks on android. In: Proceedings of the 19th NDSS (2012)

18. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R.: Xmandroid: a new android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04 (2011)
19. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: attacks and defenses. In: Proceedings of the 20th USENIX Security (2011)
20. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011). doi:10.1007/978-3-642-18178-8_30
21. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: lightweight provenance for smart phone operating systems. In: Proceedings of the 20th USENIX Security (2011)
22. Zhou, Y., Jiang, X.: Detecting passive content leaks and pollution in android applications. In: Proceedings of the 20th NDSS (2013)
23. Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., Vigna, G.: What the app is that? Deception and countermeasures in the android user interface. In: Proceedings of 36th IEEE Security and Privacy (2015)
24. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th MobiSys (2011)
25. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 19th CCS (2012)
26. Huang, H., Zhu, S., Chen, K., Liu, P.: From system services freezing to system server shutdown in android: all you need is a loop in an app. In: Proceedings of the 22nd CCS (2015)
27. Cao, C., Gao, N., Liu, P., Xiang, J.: Towards analyzing the input validation vulnerabilities associated with android system services. In: Proceedings of the 31st ACSAC (2015)
28. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of the 16th CCS (2009)
29. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: automated security certification of android (2009)
30. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th MobiSys (2012)
31. Grace, M.C., Zhou, W., Jiang, X., Sadeghi, A.-R.: Unsafe exposure analysis of mobile in-app advertisements. In: Proceedings of the Fifth WISEC (2012)
32. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the Second CODASPY (2012)
33. Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., Song, D.: Juxtapp: a scalable system for detecting code reuse among android applications. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 62–81. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37300-8_4
34. Androguard. http://code.google.com/p/androguard
35. Androbugs. http://www.androbugs.com