# Call Me Back! Attacks on System Server and System Apps in Android through Synchronous Callback

Kai Wang, Yuqing Zhang*
National Computer Network Intrusion Protection Center
University of Chinese Academy of Sciences
{wangk, zhangyq}@nipc.org.cn

Peng Liu
College of Information Sciences and Technology
The Pennsylvania State University
pliu@ist.psu.edu

## ABSTRACT

Android is the most commonly used mobile device operation system. The core of Android, the System Server (SS), is a multi-threaded process that provides most of the system services. Based on a new understanding of the security risks introduced by the callback mechanism in system services, we have discovered a general type of design flaw. A vulnerability detection tool has been designed and implemented based on static taint analysis.We applied the tool on all the 80 system services in the SS of Android 5.1.0. With its help, we have discovered six previously unknown vulnerabilities, which are further confirmed on Android 2.3.7-6.0.1. According to our analysis, about 97.3% of the entire 1.4 billion real-world Android devices are vulnerable. Our proof-of-concept attack proves that the vulnerabilities can enable a malicious app to freeze critical system functionalities or soft-reboot the system immediately. It is a neat type of denial-of-service attack. We also proved that the attacks can be conducted at mission critical moments to achieve meaningful goals, such as anti anti-virus, anti process-killer, hindering app updates or system patching. After being informed, Google confirmed our findings promptly. Several suggestions on how to use callbacks safely are also proposed to Google.

## Keywords

Mobile Security; Denial of Service; Vulnerability Detection; Synchronous Callback; Taint Analysis

## 1. INTRODUCTION

Android is an operating system for mobile devices, which is based on the Linux kernel. It occupies a large market share [7, 24] and is used in various mission critical tasks, such as vehicle-mounted systems [3], POS devices [5, 6], medical devices [1, 2, 4] and aircraft navigation [22, 28]. In order to make systems more powerful and secure, new versions of Android are released at a fast pace. One important but often unnoticed result of the system updates is that the number of system services has increased in every new version from about 50 in v2.3.7 to more than 100 in v6.0.0[1].

The number of system services is continually increased because Android needs to: 1) support emerging hardware, such as Near Field Communication (NFC) and fingerprint scanning; and 2) support new functions, such as dynamic permission authorization. It is clear that system services are critical function components in Android. They package the low level functionalities and provide essential higher level functions to apps through the Inter-Process Communication (IPC) mechanism in Android, named *Binder*. However, system services are very fragile since they provide easily accessible interfaces to third-party apps, including malicious apps. On Nexus 6 with Android 5.1.0, the System Server (SS) provides 80 Java-based services and exposes as many as 1572 interfaces. In this sense, if one failure situation occurs during the handling of one service request, the whole process may be affected. Since the SS is in fact the Android Application Framework, such failure situations can disable some core functionalities or even crash the entire system, which is clearly a single point of failure for Android system.

This paper uncovers a general type of design flaw in the SS which is caused by improper use of synchronous callback. The callback handle is received from a client process (i.e., an app). It is used to flexibly inform the client app about the handling result of a service request. A malicious app can forge a callback handle and inject it to the SS. We found that, if a synchronous callback is invoked under specific conditions inside the SS or inside a cooperator system app, vulnerability would occur. This new family of vulnerability is named as the "call me back" vulnerabilities.

Using a synchronous callback to "communicate" with untrusted apps without anticipating the worst-case situations is indeed a design flaw from the security viewpoint. In this work, we have uncovered most if not all of these worst-case situations. According to our study, in order to exploit a "call me back" vulnerability, a malicious app only needs to issue a single IPC call to the SS. The IPC sends a set of parameters to a particular service interface in the SS. For a vulnerable service interface, one of the parameters is a synchronous callback method handle. The hazard situations of the vulnerabilities are varied because the callbacks could be invoked in different contexts of the SS, or could alternatively be invoked in the context of system apps, which are the cooperators of the SS. When invoked, the malicious callback

*Corresponding author.



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



DOI: http://dx.doi.org/10.1145/2976749.2978342

[1]Summarized based on the *Genymotion* emulator, whose source code is identical to Android Open Source Project.

method can leverage two measures to conduct an attack: 1) prevent the callback method from returning, or 2) throw an exception. The attacks will result in the "freeze" of system functionalities or even the soft-reboot of the system.

According to our analysis, the attacks on the "call me back" vulnerabilities are difficult to detect and prevent. We believe the best defense method is to identify and patch the vulnerabilities as quickly as possible. However, there are several unique challenges: 1) callback handles can be injected not only as an IPC call parameter, but also as an inner field of a parameter object; 2) a callback handle can stay dormant inside the SS context for a long period of time before its invocation is triggered by some "not suspicious at all" SS operations; 3) a malicious app could try any particular combination of the IPC call parameter values; 4) any system service and any system app could be vulnerable.

We have designed and implemented a vulnerability detection tool which is based on static taint analysis. Our tool can successfully address these challenges. We applied it on all the 80 system service in the SS of Android 5.1.0 and successfully identified 6 vulnerabilities. The vulnerabilities are further confirmed on Android 2.3.7-6.0.1. It means that about 97.3% [8] of the entire 1.4 billion real-world Android devices [9] are vulnerable. The attacks prove that the vulnerabilities can enable a malicious app to freeze critical system functionalities or soft-reboot the system immediately. We also proved that our attacks can be conducted at mission critical moments to achieve very meaningful goals. Our contributions are summarized as follows:

- *New Understanding and Discovery.* Based on new understanding of the security risks introduced by the callback mechanism in system services, we have discovered a general type of design flaw which makes the Android system vulnerable to denial-of-service attacks.
- *Designing a New Vulnerability Detection Tool.* We have designed and implemented a vulnerability detection tool based on static taint analysis, which is the first work on detecting the "call me back" vulnerabilities in the SS.
- *Identifying New Vulnerabilities.* Our tool successfully analyzed 1,591 service interfaces of all the 80 system services in Android 5.1.0. We have discovered six previously unknown vulnerabilities which can affect about 97.3% of the entire 1.4 billion real-world Android devices.
- *Attack.* We have implemented several attack scenarios to show that attacks can be conducted at mission critical moments to achieve meaningful goals, such as anti anti-virus, anti process-killer, hinder app updates or system patching.
- *Defenses.* We proved it is hard to distinguish the attack from benign service requests. The best way is to detect and patch the vulnerabilities promptly. We also proposed several suggestions about how to use callbacks more safely.

# 2. BACKGROUND AND VULNERABILITY OVERVIEW

## 2.1 Android System Server

At runtime, the SS is a process. Every app is also a process. If an app wants to request a service from the SS, it will need to conduct IPC with the SS.

### 2.1.1 Binder Mechanism and Service Interfaces

Android introduces a new mechanism of IPC, namely *Binder*, into the kernel. *Binder* supports communication between an app process and the SS process following the Client-Server model. The SS leverages *Binder* to provide system services, which exposes several interfaces for the client apps.A service interface is typically a Java method inside the SS. In most cases, when an app invokes an Android API, it is actually invoking some wrapper code to conduct IPC with the target system service interface. For example, when an app invokes the `LocationManager.requestLocationUpdates()` API to register a listener for location updates, it is calling the wrapper code to send a service request to the interface `requestLocationUpdates()` in the system service named *location*.

Every system service has its own service name and interface descriptor. Using the service name, an app can query the *Service Manager* to get an instance of the system service's proxy class, which will work as the handle of the system service. The interface descriptor is an identification of service interfaces. When the app sends a request to a target service interface, it will specially declare the interface descriptor of the service in the transmitted data of IPC. And when the service receives the request, it will firstly compare the transmitted interface descriptor with its own interface descriptor. Only if the descriptors match will the request be handled by the SS.

Usually, the interface descriptor of a service is also the name of the interface-definition class for Java-based system services. This class contains an inner class named `Stub` and `Stub` also contains an inner class named `Proxy`. The service should extend `Stub` and implement the defined service interfaces. When a request arrives at the service, the implemented service interfaces will be invoked. For a client app, the handle of a service is just an object of respective `Stub.Proxy` class. `Stub.Proxy` implements the service interfaces by packaging the IPC call parameters in the transmitted data, sending the service request to the corresponding server, parsing the reply from the server and returning it to the client app.

### 2.1.2 Threads in the System Server

When a service request arrives at the SS, the *Binder* driver will start a new thread in the context of the SS to handle it. We call this kind of thread a "primary" thread. A number of system service interfaces will handle the requests with the help of an "assistant" thread to complete some time-consuming and return-value-unrelated operations. Hence, there are two kinds of thread in the server process, namely the primary threads and the assistant threads. One important difference between them is the way in which uncaught exceptions are handled.

In a primary thread, uncaught exceptions will be packaged into the reply data of the IPC. It can effectively protect the server because all the uncaught exceptions will be caught and handled. By default, when a client app receives the reply, it will automatically invoke `reply.readException()` to cause the remote exception, if exists, to be re-thrown in the context of the client app.

An assistant thread cannot re-throw the uncaught exception to the client app because it has no ability or opportunity to package the exceptions into the reply data of IPC. Actually, the uncaught exception will finally arrive at the handling code in ART/Dalvik, which maintains the VM instance for the SS. In order to recover from the bad influence of the uncaught exception, ART/Dalvik will kill the SS and soft-reboot the system.

### 2.1.3 Callback Mechanism in the System Server

Some system service interfaces receive a callback handle as one parameter. Using a callback handle, the handling result can be transmitted to the client app more flexibly. In this situation, the handling result could not only be returned by the normal reply of IPC, but also be transmitted using the received callback handle. It is a more flexible way to produce the result notification.

A callback handle is actually an object of a service's proxy class. It is a handle of a service component in an app that is waiting for the notification of the handling result from the SS. It will work as a callback method handle in the context of the SS. When it arrives at the SS through IPC, it is an object of `android.os.IBinder` class, which is the mutual ancestor class of any service's proxy class. When the callback handle is received, the SS needs to transform it into concrete class. Taking the `android.app.IInstrumentationWatcher` class as an example, the code fragment is shown as follows. We can observe that, during the transformation of an `IBinder` object, there are some validation checks (line 4). Any correct object can pass the check and be cast to the target class (line 5). However, in the event that these checks fail, the object is still forced to be treated as an instance of the target proxy class in line 6. Therefore, any received `IBinder` object will be regarded as correct.

**Example of `IBinder` Object Transformation**

```
1. public static IInstrumentationWatcher
     asInterface(android.os.IBinder obj){
2.   if (obj==null) { return null;}
3.   android.os.IInterface iin =
       obj.queryLocalInterface(DESCRIPTOR);
4.   if (((iin!=null)
       &&(iin instanceof IInstrumentationWatcher))) {
5.     return ((IInstrumentationWatcher)iin);}
6.   return new Stub.Proxy(obj);
7.}
```

Callbacks can be divided into two types:

- **Synchronous callback** is defined when the caller needs to get the return value from the callback method. Execution of the caller method will be blocked until the callback method returns.
- **Asynchronous callback** is defined when the caller does not care about the handling process of the callback. Execution of the caller method will continue (right after each invocation) without waiting for the callback to return.

## 2.2 Vulnerability Overview

All of our newly identified vulnerabilities are directly related to the IPC-based service interfaces in the SS. These interfaces receive a callback handle as an IPC call parameter. The callback can be invoked by the SS or passed to a cooperator system app to be invoked by the system app. According to the runtime context when the callback is invoked, we have identified four hazard situations:

*A*. Inside the SS:
  - *A*1. The callback is invoked in a synchronized code block of any service thread;
  - *A*2. The callback is invoked in an assistant thread without involving any synchronized block.

*B*. Inside a cooperator system app:
  - *B*1. The callback is invoked in an activity component;
  - *B*2. The callback is invoked in a service component or a broadcast receiver component.

For the situations *A*1 and *B*1, attackers can block the execution of the caller method to freeze the SS or system apps. For the situations *A*2 and *B*2, attackers can throw a carefully selected exception to the SS or system apps to crash their processes.

In our approach, we formulate the vulnerability as follows. A *vulnerability* is a controllable way (e.g., calling an interface of a system service with special parameter values) for a malicious app to let the execution of the SS or a system app reach a vulnerability point. A *vulnerability point* is a Java program statement which calls a synchronous callback method. A *vulnerability condition* is the dependent condition which determines whether a vulnerability is exploitable or not when a vulnerability point is reached.

### 2.2.1 Hazard Situation A1

If a synchronous callback is invoked in a synchronized code block in any thread inside the SS, a malicious app can implement its callback method to block the caller for a controllable duration. This results in the hazard of the freeze of a system service.

How this hazard is generated: When a service request arrives, *Binder* mechanism will start a new thread in the SS to handle it. Different threads may need to operate on the same global variable (value) in the context of the SS concurrently. Therefore, concurrency control is needed by the service threads to guarantee mutual exclusion of the multi-threaded code. The most frequently used concurrency control mechanism in the SS is based on the synchronized block mechanism from the Java library, named `java.util.concurrent`. An example is `synchronized{lock}{code}`. Threads that want to run the code in the block should acquire the lock first. The lock is accessible for only one thread in one point of time and other threads must wait for it to be released. With this mechanism, developers can ensure that: only one thread can execute the synchronized block at a time; each thread entering the synchronized block can see the effects of previous modifications; and each thread entering the synchronized block can influence later threads without conflicts.

Different synchronized blocks can be protected with the same lock. If one thread holds the lock for a long period, other threads that want to acquire this lock will be blocked and the system service will lose the ability to serve newly arrived requests. Some system services specially start a watchdog thread to monitor this kind of failure. The watchdog sets a timer-based monitor for the target lock. Once it finds that the lock cannot be acquired in a preset period, it will "bite" on the SS and force it to restart to recover from a failure state [17].

According to our new findings, some service interfaces in the SS receive a synchronous callback handle as an IPC call parameter and actually invoke it in a synchronized block. To exploit this vulnerability, attackers can implement a malicious callback method and inject its handle to these interfaces. When the callback is invoked, the attacker can block the invocation, which can freeze the SS and may finally cause the watchdog to bite on the SS.

### 2.2.2 Hazard Situation A2

If a synchronous callback is invoked in an assistant thread of the SS, the attacker can choose to reply to the service request with an exception. The exception will be thrown at the invocation statement of the callback method in the

context of the SS. If the exception cannot be handled properly, it will immediately cause the crash of the SS and the soft-reboot of the system.

For Java-based programs, *Exception* is commonly used to represent the exceptional situations. A method can inform the caller of an exceptional situation using a `throw new Exception()` statement. The caller must use *try-catch* block to catch and handle the possibly thrown exception, otherwise the code won't compile.

In this way, most kinds of *Exception* are forced to be handled explicitly by the developer, except one subclass named `RuntimeException`. `RuntimeException` can be thrown by the Java Virtual Machine (VM) in Android, i.e., DVM/ART. For instance, `NullPointerException` will be thrown when DVM/ART finds that a statement invokes a method of a `null` object. This kind of exception is more likely to be ignored by developers because it is not required to be caught explicitly. If an exception cannot be handled properly, it will finally be caught and handled by the code in DVM/ART. DVM/ART will choose to kill the process.

Our study reveals a new kind of vulnerability which is triggered when the SS invokes a malicious callback method. When invoked, the callback method chooses a subclass of *Exception*, generates one instance of the class and replies it to the caller. In the context of the SS, the exception is thrown at the invocation statement of the callback. As described in Section 2.1, the assistant threads in the SS are under the threat of uncaught exceptions. If an assistant thread invokes a malicious callback and does not handle exceptions properly, a vulnerability will occur. There are then two options for attack measures:

• **Implement a service component inside the malicious app.** The instance of a malicious service component's proxy class is leveraged as the callback handle to conduct the attack. When the callback is invoked, the service component will reply a well-chosen exception to the caller. This kind of attack has more alternatives on the replied exceptions.

• **Leverage a system service or a service component in a normal app.** The instance of a normal service's proxy class ($Service_x$) is forged as the callback handle to conduct the attack. This normal service could be a system service or a service component in an app. As described in Section 2.1, when an instance of $Service_x$'s proxy class is received by the SS as an IPC call parameter, the SS will transform it to another service, which is believed to be the right one, such as $Service_y$. However, the interface descriptors of $Service_x$ and $Service_y$ do not match. When the callback is invoked, $Service_x$ will not handle the request. It will throw a `SecurityException` back, which might not be handled properly by the callback caller.

### 2.2.3 Hazard Situation B1 & B2

Some system services expose data flow paths for a malicious app to inject malicious callback handles into system apps. A vulnerability will be triggered when the callback is invoked in the context of system apps.

One system service is not an islanding function module. There exists a synergic relationship between system services and system apps. An app often consists of four types of component, namely the *activity* (user interface), *service* (background task), *broadcast receiver* (mailbox for broadcast), and *content provider* (local database server). Some system services may interact with app components.The interactions are based on ICC (Inter-Component Communication).According to our analysis, a system service may rely on app components to perform the two following functions.

• **GUI Interaction.** Sometimes, a system service needs to interact with the device user through GUI. For example, the `usb` system service which is in charge of USB device management needs to let the user decide whether an app should get the permission to use the USB devices.

• **Functional Module.** Some system services will implement their functions by calling the service interfaces of the service components in system apps. For example, the `imms` system service will interact with the service component in the Phone app to download/send MMS.

No matter what type of work the system service assigns to a system app, it needs to inform the client app of the handling result. Usually, the system service does not work as the notifier. Instead, it passes a callback handle, which is received from the client app as an IPC call parameter, to the system app.

This seemingly neat design results in vulnerabilities. Attackers can inject a forged callback handle into a component of a system app. If a callback is invoked in an activity component of the system app (Situation *B1*), a malicious app can prevent the callback method from returning in order to freeze the GUI. And if a callback is invoked in a non-activity component of the system app (Situation *B2*), an exception can be leveraged to crash the system app process.

## 3. VULNERABILITY DETECTION TOOL

In order to discover the "call me back" vulnerabilities, we have designed a vulnerability detection tool named *KMHunter* (short for "Callback(K)-Mechanism-Hunter"). The high level idea is to implement a static taint analysis tool to identify where vulnerability exists: the IPC call parameters of system service interfaces are defined as the taint sources; and the callback invocation statements using the tainted callback handles are defined as the taint sinks.

*KMHunter*'s design is based on a widely used taint analysis tool for Android apps named *FlowDroid*. In order to apply static taint analysis on the SS, *KMHunter* is facing four challenges, which are not addressed by *FlowDroid*:

C1. Code (call-graph) dependencies of system services are more complex than apps, which makes the original class loading scheme inappropriate;

C2. Some system services utilize assistant threads to respond to service requests, which requires *KMHunter* to transform the call-graph from multi-threaded to single-threaded;

C3. ICC/IPC takes place during the handling of some service requests, which requires the generated call-graph to support ICC/IPC;

C4. A callback handle can stay dormant inside the SS context for a long period of time before its invocation is triggered by some other SS operations.

In order to address the challenges, we have modified the implementation of *FlowDroid*. The framework of *KMHunter* is shown as Figure 1 and consists of six components: Interface Analyzer is in charge of summarizing the information of service interfaces in the SS; Class Loader loads the class files as required according to specific rules; CG and CFG Generator constructs the call graph and control-flow graph centering on the target system service interface; Taint Analyzer is in charge of conducting the taint analysis based

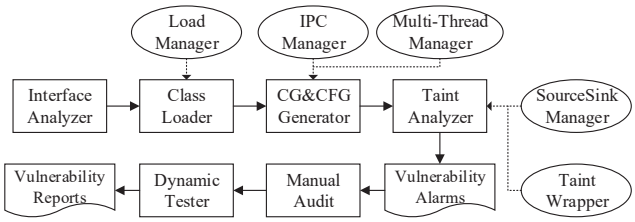
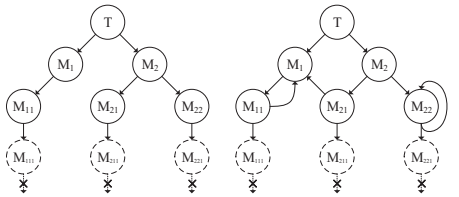



**Figure 1: Framework of KMHunter**




(a) Invocation Tree (b) Invocation Graph

**Figure 2: Examples of Class Loading Control**

on the call graph and control-flow graph; and Vulnerability Alarms can guide the Manual Audit to inspect the vulnerability conditions and develop exploit code for Dynamic Tester.

## 3.1 Class Loading Control

Code (call-graph) dependencies for system services are more complex than apps. This challenges the Class Loader components (C1). *KMHunter* will load the code of system services on demand instead of loading all the related code to address Challenge C1.

Like *FlowDroid*, the development of *KMHunter* is based on *Soot*. *Soot* is in charge of the class loading in *FlowDroid*. When an app is loaded by *Soot*, the classes of the target app will be resolved explicitly and the classes of Android APIs will be loaded as "phantom". Therefore, the class loader only needs to load the classes in the target app.

But when *Soot* is used to load the code of the SS, the class loading will be time-consuming because there is no obvious distinction between system service code and other Java libraries. *KMHunter* addresses this challenge by setting unimportant classes and methods as "phantom" to stop the loading of further related code. The key point is determining whether a method or a class is important.

One simplified example is shown in Figure 2(a). The call graph is in the form of a tree structure. The nodes represent methods and the edges represent method invocations. We define the target system service interface as **T**(arget) and other related methods as **M**. The strategy is to limit the depth of the invocation path to $N$ (which is 3 in this example) starting from **T**. Hence, methods on the third level, i.e., $M_{111}$, $M_{211}$ and $M_{221}$, will be resolved as phantom and their dependence will not be further loaded.

But in normal programs, the invocation relationship is an invocation graph instead of an invocation tree. We add some call edges in Figure 2(a) and get a call graph as shown in Figure 2(b). We can observe that, starting from $T$, the depth of method is not a constant value. For example, the depth of $M_1$ can be 1 on path $T \rightarrow M_1$ or 3 on path $T \rightarrow M_2 \rightarrow M_{21} \rightarrow M_1$, and the depth of $M_{22}$ can be $[2, +\infty)$. Under this situation, we define a function $depth_{min}(M)$ to calculate the depth of the shortest invocation path of method $M$. And if $depth_{min}(M) > N$, $M$ will be set as phantom.

For a non-phantom method, its related classes will also be loaded as normal classes. The classes include the container class of the method, the classes of parameters and local variables. For a phantom method, its related classes will also be loaded as phantom.

## 3.2 Call Graph Generation

The SS is a multi-threaded process (C2). It also interacts with other processes at runtime (C3). The CG&CFG Generator needs to address both challenges. The basic idea is to transform these complex situations into a single-threaded situation.

**Multi-Threaded SS challenge**. Commonly, a system service will start an assistant thread by two means: the *Runnable* mechanism in Java and the *Handler* mechanism in Android.

*FlowDroid* provides some basic transformations for multi-threaded situations. However, its implementation is too simple and incomplete to be used in real program analysis. We have made some improvements to complete the implementation of *FlowDroid*. The major improvement is that we take care of the *Handler* mechanism in a more subtle way. One basic usage of the *Handler* mechanism is to invoke the `Handler.sendMessage(msg)` API. When invoked, the *Handler* mechanism will dispatch the message `msg` to corresponding handling code according to the value of `msg.what`. The value of `msg.what` can vary, but the implementation code of one system service interface usually adopts only one concrete value of `msg.what` to invoke the API. If necessary, *KMHunter* will extract the code fragments, which are directly related to this concrete value, from the entire handling code for every system service interface.

**ICC/IPC challenge**. *KMHunter* needs to address the ICC/IPC challenge to analyze the taint propagation between system services and system apps. *Intent* is commonly used to perform ICC with app components. For instance, the SS can start an activity component in the system app by calling `Context.startActivity(intent)` API. The SS also can bind on the service component in a system app and perform IPC though *Binder* mechanism. No matter whether the communication is conducted by *Intent* or *Binder*, *KMHunter* will replace the IPC/ICC statements with a newly created statement. This new statement will directly invoke the corresponding handling methods in the target component. In addition, the life-circle of an app component is also considered during the call graph generation.

## 3.3 Defining the Taint Sources and Sinks

By neatly defining the sources and sinks, we can address the challenge of the dormant callback handle (C4).

During the taint propagation in the SS, we define the set of tainted objects as *TOS* (Tainted Object Set). The tainted state of the target system service can be divided into two types: temporarily tainted and durably tainted.

**The SS is temporarily tainted** if every $t \in TOS$ is a local variable. The tainted objects will be destroyed when the handling of one service request finishes. Therefore, the taint analysis just needs to consider the implementation code of one system service interface. We define the source and sink of the taint analysis as:

$$\texttt{Source}_{\texttt{inject}} =< \texttt{SSI}, \texttt{i}, \texttt{p} >,$$
$$\texttt{Sink}_{\texttt{crash}} =< \texttt{condition}, \texttt{callback\_statement}, \texttt{t} >$$

In the definition of $\texttt{Source}_{\texttt{inject}}$, `i` is the index of the IPC call parameter `p` of the service interface `SSI`. And in the definition of $\texttt{Sink}_{\texttt{crash}}$, `t` means the tainted data which will be used by the `callback_statement` under the given `condition`.

**The SS is durably tainted** if there exists at least one $t \in TOS$ that is a global variable. The tainted object can stay dormant inside the SS context for a long period of time before its invocation is triggered by other SS operations. *KMHunter* has to consider the cooperative relation of service interfaces and other trigger points such as registered broadcast receivers. The strategy is to separate the taint analysis into two steps. The definitions of source and sink in the two steps are:

`Step 1:`

$$\texttt{Source}_{\texttt{inject}} =< \texttt{SSI}, \texttt{i}, \texttt{p} >,$$
$$\texttt{Sink}_{\texttt{global}} =< \texttt{none\_condition}, \texttt{derived}(\texttt{SSO.x}, \texttt{t}), \texttt{t} >$$

`Step 2:`

$$\texttt{Source}_{\texttt{global}} =< \texttt{SSO}, \texttt{x} >,$$
$$\texttt{Sink}_{\texttt{crash}} =< \texttt{condition}, \texttt{callback\_statement}, \texttt{t} >$$

In the definition of $\texttt{Sink}_{\texttt{global}}$ in Step 1, a statement is regarded as a sink if a global value, namely `SSO.x`, is derived from a tainted valuable `t`. In the definition of $\texttt{Source}_{\texttt{global}}$ in Step 2, a global variable `SSO.x` is regarded as a source if it is tainted in Step 1.

## 3.4 Human Intelligence

Currently, *KMHunter* still needs human intelligence to manually craft the test cases. The taint analysis results do not directly enable automated generation of dynamic test cases. Another limitation is that our tool does not guarantee completeness (i.e., identifying all the vulnerability conditions). It ignores the code statements which do not operate on the tainted data; however, these code statements could relate to the vulnerability conditions, such as checking the permissions of the client app.

# 4. VULNERABILITY DETECTING RESULT

## 4.1 Result Overview

We applied *KMHunter* on the code of Android 5.1.0. According to the Interface Analyzer component in *KMHunter*, the SS contains 80 Java-based system services. These services expose 1592 service interfaces. *KMHunter* has successfully analyzed 1591 of them (the only failure is because that the class loading process failed). Vulnerability alarms sounded for eleven service interfaces. After manual inspection, we identified six vulnerabilities in nine service interfaces. The vulnerabilities are listed in Table 1. All the identified vulnerabilities have been further tested on Android 2.3.7-6.0.1[2]. The test result indicates that the "call me back" vulnerabilities exist widely.

There are two false positives. The first false positive is an interface which is provided by the window system service. It is protected by a permission which could not be acquired by third-party apps. The other false positive is a service interface which is provided by the display system service. This interface checks the validation of the received callback handle. Only the instance of a specific system service's proxy class will be accepted. This system service is media_projection. There may be false negatives. But they cannot be analyzed since there is no ground truth; we are the first to reveal the "call me back" vulnerabilities. False negatives may exist because of our class loading strategy. Some methods will not be loaded because their shortest invocation paths are deeper than a threshold. Although the analyzed call graph is not complete, our analysis managed to cover most of them. For example, the analyzed call graph of the setLastChosenActivity() interface in PMS contains 10,595 edges.

[2]v2.3.7-v5.0.0 on *Genymotion* emulator, v5.1.0 on Nexus 6 and v6.0.1 on Nexus 6p.

**Table 1: Newly Discovered Vulnerabilities**

| *Id* | *Service* | *Service Interface* | *Versions* |
|---|---|---|---|
| Vul#1 | activity | startInstrumentation | 4.2 - 6.0.1 |
| Vul#2 | location | requestLocationUpdates | 4.2 - 6.0.1 |
| Vul#3 | mount | registerListener | ? - 5.1.0* |
| Vul#4 | package | freeStorage | 2.3.7 - 5.1.0 |
| Vul#5 | usb | requestDevice-(Accessory)Permission | 4.1.1 - 6.0.1 |
| Vul#6 | imms | send(Stored)Message, downloadMessage | 5.0.0 - 6.0.1 |

**: Tests of mount service need real devices. Hence, it is only tested on v5.1.0 and v6.0.1.*

## 4.2 Vulnerability Details

### 4.2.1 Vulnerability in Activity Manager Service

AMS (*Activity Manager Service*) is in charge of interactions with overall activities running in Android. The vulnerable service interface is `startInstrumentation()`. Apps can call it to start an instrumentation component of a given app. This interface receives a callback handle, named `watcher`, as an IPC call parameter. If a failure situation happens while starting target instrumentation, this callback will be invoked to inform the client app. The class of `watcher` is `android.app.IInstrumentationWatcher`. It contains two synchronous callback methods, named `instrumentationStatus()` and `instrumentationFinished()`.

The vulnerable code is shown in Fragment 1 of Appendix A. We can observe that the code statements of `startInstrumentation()` enter the `synchronized(AMS.this)` block. In the synchronized block, if a failure situation (such as wrong instrument info) occurs, the method named `reportStartInstrumentationFailure()` will be called (line 16466). This method receives the `watcher` object as an actual parameter. Code in this method invokes the callback method `watcher.instrumentationStatus()` (line 16525). This is a typical vulnerability in hazard situation $A1$.

This vulnerability exists in Android 4.2-6.0.1. The vulnerable service interface does not check any permission of the client app. Hence, malicious apps can exploit this vulnerability without restrictions.

### 4.2.2 Vulnerability in Location Manager Service

LMS (Location Manager Service) manages location providers and issues location updates and alerts. Apps can invoke a service interface of LMS named `requestLocationUpdates()` to register a callback for location updates. This interface receives a parameter of `android.app.PendingIntent`

class named `intent`. The `intent` object is not a callback handle. But it carries a member variable named `mTarget`, which is a callback handle. The class of `mTarget` is `android.content.IIntentSender`. In this class, only the callback method named `send()` is defined as synchronous. If LMS invokes `intent.send()`, it is actually invoking the `intent.mTarget.send()` method.

The vulnerable code is in Fragment 2 of Appendix A. The callback is invoked (line 854) in a synchronized block (line 1574). When this vulnerability is exploited, the system would get into hazard situation $A1$.

This vulnerability exists in Android 4.2-6.0.1. The vulnerable service interface requires the client app to hold the `access_coarse(fine)_location` permissions. These permissions are all acquirable for a malicious app.

### 4.2.3 Vulnerability in Mount Service

MS (Mount Service) is responsible for various storage media. It connects to vold to watch for and manage dynamically added storage, such as SDcards and USB mass storage.

The service interface in MS, named `registerListener()`, can be called to register a listener (i.e., callback) on the state changes of storage or USB mass storage. The class of this listener is `android.os.storage.IMountServiceListener`. It defines two synchronous callback methods. As shown in Fragment 3 of Appendix A, the tainted callback handle propagates into the global variable named `mListeners` (line 1557). MS also registers a broadcast receiver to listen on the intent, which is broadcasted when the state of USB storage is changed (line 1498). When MS receives the intent, it will make callbacks based on every element in `mListeners` (line 1247, 1249). During this period, it is necessary to ensure the mutual exclusiveness of `mListeners`. Therefore, the related code is in a synchronized block. When this vulnerability is exploited, the system would get into hazard situation $A1$.

This vulnerability exists in Android 5.1.0 and no longer exists in Android 6.0.1, because Android has changed the type of the callback from synchronous to asynchronous. The vulnerable service interface has no permission requirement on the client app.

### 4.2.4 Vulnerability in Package Manager Service

PMS (Package Manager Service) keeps track of all those .apks everywhere. The vulnerable service interface in PMS is `freeStorage()`. It is in charge of clearing the cache of given volume Universally Unique Identifier (UUID). The related code is shown in Fragment 4 of Appendix A. Cache clearing is a time-consuming work. PMS carries out this work in an assistant thread leveraging the *Handler* mechanism (line 2196). Code from the assistant thread is implemented inside the `freeStorage()` method. Hence, it can operate the IPC call parameter named `pi`. The class of `pi` is `android.content.IntentSender`. Similar to `PendingInent`, this class also contains a member variable, which is a callback handle. The member variable is named as `mTarget`. The class of `mTarget` is also `IIntentSender`. This class only defines one synchronous callback method named `send()`. After the cleaning work finishes, the assistant thread will invoke the `pi.sendIntent()` method (line 2207). Actually, it is invoking the callback method of `pi.mTarget.send()`.

In line 2206, PMS tries to check on the validity of `pi`. However, it could not validate the identity of the service represented by `pi.mTarget`. What's more, PMS can only handle one type of exception, namely `SendIntentException` in line 2212. Hence, we could not expect it to survive any other types of exception that may be replied from the callback method. When this vulnerability is exploited, the system would get into hazard situation $A2$.

The vulnerable service interface is protected with permission `clear_app_cache`. This permission is acquirable for third party apps before Android 5.1.0, but not in Android 6.0.1. Therefore, although the vulnerability still exists in Android 6.0.1, it is only exploitable in v2.3.7-v5.1.0.

### 4.2.5 Vulnerability in SystemUI App

US (USB Service) manages all USB-related state, including both host and device support. The `requestAccessoryPermission()` interface in US can be invoked by a client app to request the permission of USB accessories. When it is invoked, US will start an activity in the SystemUI app to let the device user decide whether or not the permission should be granted. This interface receives a callback handle as an IPC call parameter. The parameter is named as `pi`, whose class is `android.os.PendingIntent`. It will be passed to the activity component in SystemUI. SystemUI uses it to inform the client app of the decision of the device user. As described in Section 4.2.2, when `pi.send()` is invoked by the activity component in SystemUI, a synchronous callback method is actually invoked.

The related code is shown in Fragments 5.1, 5.2 and 5.3 of Appendix A. We can see that US starts the activity component in SystemUI and passes the callback handle to it by the `intent` object (line 1045 in 5.2). The target activity component acquires the callback handle in the `onCreate()` method (line 68 in 5.3) and invokes it when the activity is to be destroyed (line 146 in 5.3). Malicious services can block the code of the activity component by preventing the callback method from returning until it is too late. When this vulnerability is exploited, the system would get into hazard situation $B1$.

This vulnerability exists in Android 4.1.1-6.0.1. The vulnerable service interface is not protected with any permission. Another interface of US named `requestDevicePermission()` also has this vulnerability.

### 4.2.6 Vulnerability in Phone App

The imms system service bridges the public SMS/MMS APIs with the service interfaces of the MmsService component in the Phone app. This kind of design can protect the integrity of the SS. However, it leaves the Phone app at risk of attack.

The vulnerable service interfaces in imms are `downloadMessage()`, `sendMessage()` and `sendStoredMessage()`. They all receive an IPC call parameter whose class is `PendingIntent`. As described in Section 4.2.2, the object of the `PendingIntent` class contains a callback handle as its member variable.

We take the `downloadMessage()` interface as an example. The related code is shown in Fragments 6.1, 6.2 and 6.3 of Appendix A. Line 253 in Fragment 6.1 shows that imms calls the corresponding service interface of the MmsService component. The received callback handle is also passed to MmsService. This callback handle is used to inform the client app about the result of sending/downloading MMS (line 230 in Fragment 6.3). The invocation of this callback is located in an assistant thread of MmsService (line 427 in

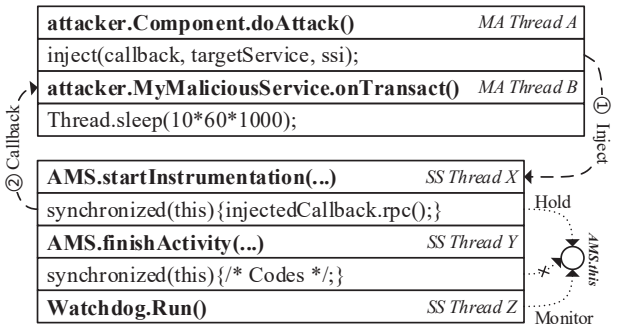



**Figure 3: One Attack on ActivityManagerService**

Fragment 6.2). The invocation statement is not protected by any *try-catch* block to handle exceptions. When this vulnerability is exploited, the system would get into hazard situation $B2$.

This vulnerability exists in Android 5.0.0-6.0.1. The exploit code needs the permission of `receive(send)_mms`, which is acquirable for a malicious app.

# 5. PROOF-OF-CONCEPT ATTACKS

## 5.1 Basic Attacks and Hazards

We present the basic exploitation of the new vulnerabilities here. The attacks are implemented by directly invoking the vulnerable service interfaces of the SS instead of some APIs [17]. Therefore, they are more direct and flexible.

### 5.1.1 System Service Freeze

In order to freeze a system service, the malicious app needs to implement a callback method which prevents the returning of callback for a long period of time.

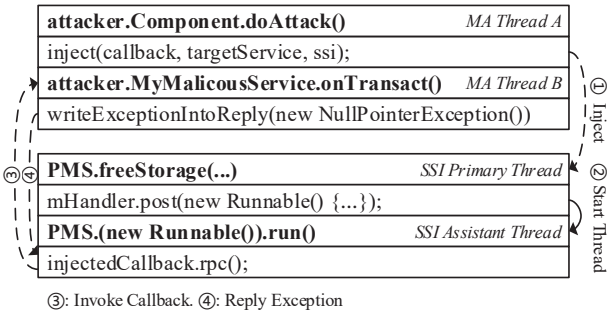
Taking $Vul\#1$ as example, the logical relationship of the attack is shown in Figure 3. The instance of MyMaliciousService's proxy class is forged as an IPC call parameter. It is passed to the service interface `startInstrumentation()` of AMS. Then AMS starts *thread X* to handle the request. When *thread X* invokes the callback method, the `onTransact()` method in the MyMaliciousService component will be in charge of the handling. The attack code in MyMaliciousService is very simple in that it sleeps for ten minutes. Therefore, *thread X* becomes blocked. *Thread Y* is another thread which is started by another service request. It also wants to acquire the `AMS.this` lock, but it has to wait until *thread X* releases it. After a period of time, the `watchdog` thread in AMS will kill the process which provides AMS, namely the SS. The crash of the SS will result in the soft-reboot of the system.

$Vul\#2$ is similar to $Vul\#1$, except that it requires specific permission. $Vul\#3$ is a little different, because it is not triggered immediately after the callback handle is injected. Hence, the attacks on $Vul\#3$ are more latent than others.

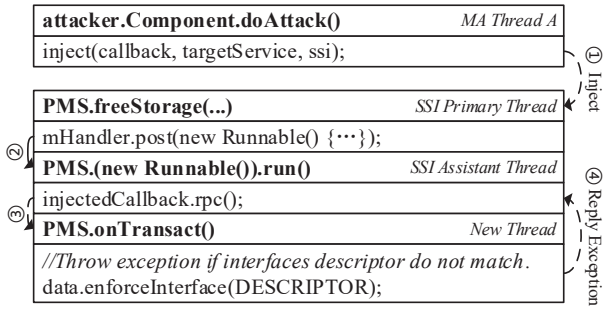
### 5.1.2 System Soft-Reboot

To soft-reboot the system in a timely manner, a malicious app must find a way to make the SS throw an exception. The attacks are varied because there are two possible attack measures.

We take the logical relationship of the attacks on $Vul\#4$ as an example. In Figure 4(a), a malicious service is used to conduct the attack. When the callback method is invoked, the malicious service selects one exception, namely `NullPointerException`, and throws it back. In Figure 4(b), PMS is attacked by the exception thrown by itself. Firstly, the instance of PMS's proxy class is forged as a callback handle. It is injected to the `freeStorage()` interface of PMS as an IPC call parameter. PMS invokes this callback in one of its assistant threads. It believes the callee is a service whose interfaces are defined by the `IIntentSender` class. However, the real callee is itself, whose interfaces are defined by the `android.content.pm.IPackageManager` class. As the callee, PMS starts a new thread. Code in this new thread finds that the interface descriptors do not match. Then a `java.lang.SecurityException` is thrown from the new thread back to the assistant thread in PMS. It is able to cause the crash of the process which provides PMS, namely the SS.




(a) Leverage a Malicious Service Component to Attack




(b) Induce PackageManagerService to Attack Itself

**Figure 4: Two Attacks on PackageManagerService**

### 5.1.3 System Application Freeze and Crash

Even if the victim of an attack is a system app, the attack hazard also will seriously threaten some critical system functionalities.

When $Vul\#5$ is exploited, SystemUI will raise an activity component on the screen. No matter how the device user interacts with the activity, the callback method in the malicious app will be invoked. The attacker can block the code in this activity to make SystemUI unresponsive. $Vul\#6$ exists in the Phone app. The Phone app is in charge of the cellular networks. Attacks on this app can cause it to crash and disable all the functionalities of the cellular networks.

The logical relationships of the attacks on $Vul\#5$ and

$Vul\#6$ are similar to Figure 3 and 4. The only difference is that the callback handle is passed from a system service to a vulnerable app. Therefore, we do not provide detailed figures here. Although the victim apps can be restarted quickly, the malicious app can continuously trigger the vulnerabilities to conduct repetitive attacks.

## 5.2 Representative Attack Scenarios

Indiscriminate attacks are meaningless. If an attacker wants to maximize the attack effect, it should choose the best time to exploit the vulnerabilities. Previous work [14] has shown that it is possible to peek into the apps through a UI state inference attack. Therefore, attackers can trigger the vulnerabilities when the system is conducting critical tasks. For example:

$Vul\#2$ can block the location updates of map apps, which will disable the navigation functions;

$Vul\#3$ can make the SDcard unusable, which may contain installed apps and app data;

$Vul\#5$ can repetitively block the SystemUI to prevent the user from pressing specific buttons;

$Vul\#6$ can disable the cellular networks.

$Vul\#1$ and $Vul\#4$ are the two most significant among the new vulnerabilities. We have leveraged some side channels to monitor the system and designed four attack scenarios exploiting these two vulnerabilities. A video of the attacks has been uploaded onto https://youtu.be/w9BMZdvZZec.

### 5.2.1 Anti Process Killer

**Scenario.** In order to conduct a timely attack, the process of a malicious app needs to stay alive in the background. However, a background process is unwelcome and may be killed irregularly.

**Design.** Android supports an app to run its services in different processes. If an app is forced to stop, all of its processes will be killed successively. In our design, the malicious app registers *N* services in *N* different processes. Each service listens to the death of the other *N-1* services by implementing the *ServiceConnection.onServiceDisconnected()* method. Code in this method will exploit the $Vul\#4$, which can immediately cause the soft-reboot of the system. The malicious app can register a broadcast receiver component to listen to the `boot_completed` intent. Then it can start its services again to stay alive.

**Result.** We installed the PoC app on Android 5.1.0, Nexus 6. This app registers 10 services in 10 processes. We tried killing the processes of the app using the third party app *360 Mobile Safe* and the system app *Settings*. The test result proves that the PoC app can function as an anti process killer.

### 5.2.2 Anti Anti-Virus

**Scenario.** When a device user encounters too many instances of system crash and functional failure, he/she may suspect that the device is infected by malicious apps. Most user will choose an anti-virus app to make a security inspection. The malicious apps need to hinder this to ensure their survival.

**Design.** A malicious app can utilize many characteristics to detect whether an anti-virus app is scanning the apps. For example, the scanning significantly increases the memory usages of the anti-virus app. Some anti-virus apps start new processes to do the scanning. The malicious app only needs a list of the process names of the anti-virus apps. It can carry out a real-time monitor on the listed processes to detect the virus scanning.

**Result.** We installed the PoC app on Android 5.1.0, Nexus 6. *360 Mobile Safe* was selected as the attack target. According to our study, it starts two processes, namely *scan* and *engine* when scanning the installed apps. The PoC app checks whether these processes are alive every one second. The vulnerability $Vul\#1$ was selected to conduct the attack. Exploiting $Vul\#1$ can make the device unresponsive to any GUI operation. It gives the impression that the virus scanning consumes too many computing resources. The scanning also will be blocked until the system soft-reboots.

### 5.2.3 Hindering Critical Application Patching

**Scenario.** Due to the evolving nature of mobile systems, apps have to update for vulnerability patching. Malicious apps need to hinder the patching of critical apps, which will render the vulnerable apps unpatched.

**Design.** The app updating is conducted by PMS. It can be divided into three sequential subtasks: removing the original app, adding a new app and configuring the new app. When each subtask finishes, a broadcast will be send with corresponding action tags. The malicious app can check the existence of the target apps by calling `PackageManager.getApplicationInfo()` frequently. This helps to monitor the removal of the old version. Once the target app no longer exists, the malicious app can hinder the installation of the new version or soft-reboot the system immediately. When the system finishes the soft-reboot, PMS will roll back the unfinished update task to ensure its atomicity.

**Result.** The PoC app was deployed on Android 5.1.0 of Nexus 6 to prevent the update of *360 Mobile Safe*. It checks the existence of the target app every 5 milliseconds. The $Vul\#1$ was leveraged to conduct the attack. It led to the freeze of AMS. Hence, PMS could not utilize AMS to send the broadcast after the old version app was removed. After about 62 seconds of blocking, the whole system crashed. The update rolled back after reboot.

### 5.2.4 Hindering System Updating

**Scenario.** Android is evolving rapidly. The newly found vulnerabilities are patched quickly in the new versions. If an attacker can find a way to hinder the update of the system, it will ensure the system remains vulnerable forever.

**Design and Implementation.** For ordinary users, the most frequently used measure to update the system is through an OTA (Over-The-Air) update. The OTA update needs to download the update files from the network server and store them in local devices. To monitor the OTA update, the malicious app can scan the file system to detect the downloaded update files. It also could monitor the process in charge of the OTA update by checking the amount of received bytes from the internet. When the malicious app finds the system is updating, it will have many options. It can exploit $Vul\#1$ or $Vul\#4$ to freeze and soft-reboot the system. It also can exploit $Vul\#6$ to crash the Phone app in order to hinder the download through the mobile network.

**Result.** We deployed the PoC app on Nexus 6, which needed to update from 5.1.0 to 5.1.1. The size of the update package was about 110MB. The *Google Mobile Service* (GMS) app was in charge of the OTA update. The PoC app monitored the number of bytes received by the GMS

app by invoking the `TrafficStats.getUidRxBytes()` API. When the number surpassed 10 MB, $Vul\#4$ was triggered. It soft-rebooted the system immediately. The test result shows that the PoC app can successfully prevent the OTA update.

# 6. DEFENSE APPROACHES

Attacks on the new vulnerabilities are difficult to prevent due to the reasons:

1). in the aspect of the *Binder* driver, a malicious service request is indistinguishable from benign requests;
2). in the aspect of the SS, a received callback handle is unverifiable to prevent attack;
3). in the aspect of dynamic monitoring, when the callback is invoked, the malicious service component could choose to attack under certain timing to avoid exposing its aggressive behavior;
4). in the aspect of static audit, the attack code can be developed with both Java and C/C++ languages.

Therefore, the best way to defend against the attacks is to identify and patch the vulnerabilities as quickly as possible. To patch these vulnerabilities, we propose the following suggestions for the developers of system services.

*For some callbacks, the callers have no need to wait for the reply. These callbacks must be declared as asynchronous.* AMS receives a callback handle whose interfaces are defined in the `IInstrumentationWatcher` class. MS receives a callback handle whose interfaces are defined in the `IMountServiceListener` class. These two system services are vulnerable to the "call me back" vulnerability. After an in-depth study on them, we find that all the return types of their interfaces are `void`. There is no need to wait for the return of these callback methods at all. Therefore, there is a very convenient way to patch these vulnerabilities, which is to change the type of these callbacks from synchronous to asynchronous.

*For some callbacks, the callers have to wait for the reply. Their invocation statements must not be in a synchronized block. And a caller should invoke a callback in a try-catch block which can handle all possible exceptions.* Four of the new vulnerabilities are caused by receiving a parameter that contains a callback handler as a member variable. Actually the class of the callback handles is the same, which is the `IIntentSender` class. This class only declares one callback method named `send()`. This callback method returns a value which indicates whether or not an intent is sent successfully. Hence, it has to be a synchronous callback. Under this circumstance, the developers should not invoke the callback in a synchronized block. He/she should try to catch and handle all the possible exceptions using *try-catch* blocks.

From another perspective, although the developers can anticipate the worst-case situations before implementing a new service interface which accepts a callback handle as a parameter, a more fundamental security design question is whether a system service developer should ever use a synchronous callback to "communicate" with untrusted apps. There are some other ways to "communicate" with an untrusted app, such as socket, pipe and shared memory. No matter which way is chosen, the key point is whether the communication should be synchronous. We insist that one design principle of system services is that it should never communication with untrusted apps in a synchronous way.

# 7. RELATED WORK

Vulnerabilities in system services have been explored by several previous studies. A general design trait in the concurrency control mechanism of the system services has been discovered by [17]. This new kind of vulnerability is named as ASV, which makes the Android system vulnerable to Denial-of-Service attacks. The hazard situation of ASV is similar to the situation $A1$ in this paper (see Section 2.2). The limitation of [17] is that: 1) they believe that the attack surfaces are the APIs which wrap the invocation code of vulnerable service interfaces instead of the interfaces themselves; 2) the exploitation code needs to repeatedly invoke the attack surfaces or register unusually large numbers of resources, which is easy to detect and prevent; 3) only AMS and PMS are found to be vulnerable and the attacks only can freeze parts of their functionalities immediately. Our work have found four hazard situations, which makes the attacks more various. Our attacks only need to invoke the vulnerable service interfaces once. They can freeze critical target functionalities or soft-reboot the system immediately. Therefore, our attacks are more flexible to be leveraged in different attack scenarios (see in Section 5.2).

Some other works designed fuzzing tools targeted on the system services in Android [13, 16, 19]. They leveraged the idea that *Binder* provides a very convenient way to inject test cases into system services. Fuzzing tests are easy to implement and effective on vulnerability detection. However, there are two main challenges to designing a perfect fuzzing tool for the system services in Android. Firstly, the strategy of test data generation can seriously affect the effect and efficiency of a fuzzing test. The designer cannot guarantee that the test cases cover all the execution paths of the target system service. Secondly, it is hard to define the abnormal behavior for every system service interface. Therefore, the false negative rate is highly dependent on the definitions of abnormal behaviors. If a designer wants to cover all the possible failure situations, he/she has to manually inspect the source code of every system service. This is heavy and clumsy work. Our static taint analysis tool can easily cover all the execution paths. Since we have clearly characterized the feature of the "call me back" vulnerabilities, we do not have to blindly generate test cases and struggle to monitor all the possible failure situations of the system services.

Taint analysis has been widely applied to different platforms for different purposes [11, 12, 21, 25, 26, 27]. For Android, the main usage of taint analysis is to detect privacy leakages of the apps. They can be divided into dynamic taint analysis [15, 23] and static taint analysis [10, 18, 20, 29, 30]. *TaintDroid* is a dynamic taint analysis tool for real-time privacy monitoring on Android apps [15]. It uses variable-level tracking within the VM interpreter while the target app is running. It is very effective on real-time monitoring but cannot guarantee that all the execution paths have been covered. Other researchers focus on the design of static taint analysis tools. *FlowDroid* is an excellent framework for static taint analysis. It can provide precise context, flow, field, and object-sensitive and lifecycle-aware taint analysis for Android apps [10]. However, it is not entirely applicable for vulnerability detection on the SS, as we have described in Section 3. Our work is the first to introduce static taint analysis on the code audit and vulnerability detection targeted on the SS in Android.

# 8. CONCLUSIONS

Based on a new understanding of the security risks introduced by the callback mechanism in system services, we have discovered a general type of design flaw. It reveals a new kind of vulnerability in system services and system apps. We have designed and implemented a vulnerability detection tool based on static taint analysis.Our tool has successfully analyzed all the 80 system services in Android 5.1.0. With its help, we discovered six previously unknown vulnerabilities, which are further confirmed on Android 2.3.7-6.0.1. These vulnerabilities affect about 97.3% of the entire 1.4 billion real-world Android devices. We crafted several PoC apps and illustrated the serious attack hazards from the freeze of critical functionalities to the soft-reboot of the system. We also designed several attack scenarios and proved that the vulnerabilities can enable malicious apps to attack the system at mission critical moments, such as system updating and virus scanning. The newly found vulnerabilities have been reported to Google and Google confirmed them promptly. Some suggestions are also proposed for the developers of system services and apps to patch and prevent this new kind of vulnerability.

# 9. ACKNOWLEDGMENTS

Kai Wang and Yuqing Zhang were supported by the National Natural Science Foundation of China (61272481, 61572460), the National Key Research and Development Project (2016YFB0800703), the National Information Security Special Projects of National Development and the Reform Commission of China [(2012)1424]. Peng Liu was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1422594, and NSF CNS-1505664.

# 10. REFERENCES

[1] 27 million doctors' mobile devices at high risk of malware | ITProPortal.com. http://goo.gl/BJs5Mu.

[2] Android and RTOS together: The dynamic duo for today's medical devices - embedded computing design. http://goo.gl/StURzu.

[3] Android auto. https://www.android.com/auto/.

[4] Android OS for smart medical equipment, developing embedded medical devices | hughes systique. http://goo.gl/aO0NFk.

[5] Android point of sale | android POS restaurants, cafes, bars | tablet POS. http://www.posandro.com/.

[6] The best android POS of 2016 | top ten reviews. goo.gl/9xykVH.

[7] Gartner says worldwide smartphone sales grew 9.7 percent in fourth quarter of 2015. http://goo.gl/M0ZwSk.

[8] Google says there are now 1.4 billion active android devices worldwide. http://goo.gl/utHxO8.

[9] Lollipop is now the most-used version of android, marshmallow up to 2.3 percent. http://goo.gl/Q598DH.

[10] 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269. ACM.

[11] J. Bell and G. Kaiser. Dynamic taint tracking for java with phosphor (demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 409–413. ACM.

[12] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 3–8. ACM.

[13] C. Cao, N. Gao, P. Liu, and J. Xiang. Towards analyzing the input validation vulnerabilities associated with android system services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 361–370. ACM.

[14] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1037–1052, San Diego, CA, Aug. 2014. USENIX Association.

[15] W. Enck, P. Gilbert, 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. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407. USENIX Association.

[16] G. Gong. Fuzzing android system services by binder call to escalate privilege. https://www.blackhat.com/us-15/briefings.html.

[17] H. Huang, S. Zhu, K. Chen, and P. Liu. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1236–1247. ACM.

[18] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 106–117. ACM.

[19] W. Kai, Z. Yuqing, L. Qixu, and F. Dan. A fuzzing test for dynamic vulnerability detection on android binder mechanism. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 709–710.

[20] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6. ACM.

[21] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium*.

[22] D. Lundberg, B. Farinholt, E. Sullivan, R. Mast, S. Checkoway, S. Savage, A. C. Snoeren, and K. Levchenko. On the security of mobile cockpit information systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 633–645. ACM.

[23] J. Paupore, E. Fernandes, A. Prakash, S. Roy, and X. Ou. Practical always-on taint tracking on mobile devices. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 29–29.

[24] S. V. President and BCG. Android OS smartphone market share worldwide 2009-2015 | statistic. http://goo.gl/9mI3Qw.

[25] A. Rountev, M. Sharp, and G. Xu. IDE dataflow analysis in the presence of large object-oriented libraries. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 53–68. Springer-Verlag.

[26] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4f: Taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1053–1068.

[27] Z. Wei and D. Lie. LazyTainter: Memory-efficient taint tracking in managed runtimes. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 27–38. ACM.

[28] R. Wilkers. Northrop to demo DARPA navigation system on android; charles volk comments. http://goo.gl/dLmhXN.

[29] Z. Yang and M. Yang. LeakMiner: Detect information leakage on android with static taint analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering*, WCSE '12, pages 101–104. IEEE Computer Society.

[30] Z. Zhao and F. C. Colon Osono. "TrustDroid$^{TM}$": Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking. In *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software (MALWARE)*, MALWARE '12, pages 135–143. IEEE Computer Society.

# APPENDIX

# A. VULNERABLE CODE

The source code is excerpted from the AOSP v5.1.0_r3. Some unrelated code is omitted.Some unrelated parameters are also replaced with dots.

**Fragment 1. Code in ActivityManagerService**

```
16442. public boolean startInstrumentation(...,
         IInstrumentationWatcher watcher, ...){
16454.   synchronized(this) {
16466.     reportStartInstrumentationFailure(
             watcher,...);
16506. }}
16517. private void reportStartInstrumentationFailure(
         IInstrumentationWatcher watcher, ...){
16525.   watcher.instrumentationStatus(...);
16530. }
```

**Fragment 2. Code in LocationManagerService**

```
584.  private final class Receiver{ //Inner Class
591.    final PendingIntent mPendingIntent;
825.    public boolean callProviderEnabledLocked(...){
854.      mPendingIntent.send(...));}}
1548. public void requestLocationUpdates(...,
        PendingIntent intent,...){
1574.   synchronized (mLock) {
          //recevier.mPendingIntent = intent
1577.     requestLocationUpdatesLocked(
            ..., receiver, ...);}}
1584. private void requestLocationUpdatesLocked(...,
        Receiver receiver, ...) {
1612.   receiver.callProviderEnabledLocked(...)
1613. }
```

**Fragment 3. Code in Code in MountService**

```
119.  class MountService{
236.    ArrayList mListeners = new ArrayList<...>();
690.    private final BroadcastReceiver mUsbReceiver =
          new BroadcastReceiver() {
692.      public void onReceive(...) {
695.        notifyShareAvailabilityChange(available);
697.    }};
1243.   private void notifyShareAvailabilityChange(...) {
1244.     synchronized (mListeners) {
            //i âĹĹ [0,mListeners).size())
1247.       bl = mListeners.get(i);
1249.       bl.mListener.
              onUsbMassStorageConnectionChanged(...);
1290.   }}
1478.   public MountService(...) {
1498.     mContext.registerReceiver(mUsbReceiver, ...);
1541.   }
1552.   public void registerListener( //target SSI
          IMountServiceListener listener) {
1557.     mListeners.add(bl); //bl.mListener = listener
3138. }}
```

**Fragment 4. Code in PackageManagerService**

```
2192. public void freeStorage(..., IntentSender pi) {
2196.   mHandler.post(new Runnable() {
2197.     public void run() {
2206.       if(pi != null) {
2207.         try { pi.sendIntent(...);
2212.         }catch(SendIntentException e1){...}}}});}
```

**Fragment 5.1. Code in UsbService**

```
2192. public void freeStorage(..., IntentSender pi) {
2196.   mHandler.post(new Runnable() {
2197.     public void run() {
2206.       if(pi != null) {
2207.         try { pi.sendIntent(...);
2212.         }catch(SendIntentException e1){...}}}});}
```

**Fragment 5.2. Code in UsbSettingsManage**

```
1023. private void requestPermissionDialog(
        Intent intent, ..., PendingIntent pi){
1038.   intent.setClassName(``com.android.systemui'',
1039.     ``.usb.UsbPermissionActivit'');
1041.   intent.putExtra(Intent.EXTRA_INTENT, pi);
1045.   mUserContext.startActivityAsUser(intent, ...);
1051. }
```

**Fragment 5.3. Code in UsbPermissionActivity**

```
46.  public class UsbPermissionActivity{
55.    private PendingIntent mPendingIntent;
62.    public void onCreate(...) {
68.      mPendingIntent = (PendingIntent) intent.
           getParcelableExtra(Intent.EXTRA_INTENT);
115. }
118. public void onDestroy(){
146.   mPendingIntent.send(this, 0, intent);
175. }}
```

**Fragment 6.1. Code in MmsServiceBroker**

```
239. public void downloadMessage(...,
       PendingIntent downloadedIntent){
253.   getServiceGuarded().//return MmsService proxy
         downloadMessage(..., downloadedIntent);
255. }
```

**Fragment 6.2. Code in MmsService**

```
115. private class RequestQueue extends Handler {
121.   public void handleMessage(Message msg) {
125.     request.execute(...); //request = msg.obj
128. }}
204. public void downloadMessage(...,
       PendingIntent downloadedIntent)
       //request.mDownloadedIntent = downloadedIntent
219.   addSimRequest(request);
221. }
367. RequestQueue[] mRunningRequestQueues;
417. private void addToRunningRequestQueueSynchronized(
       MmsRequest request){
425.   final Message message = Message.obtain();
426.   message.obj=request;
427.   mRunningRequestQueues[queue].sendMessage(message);}
```

**Fragment 6.3. Code in MmsRequest**

```
131. public void execute(...) {
197.   processResult(...);}
208. public void processResult(...) {
230.   pendingIntent.send(...);
237. }
```