Exception Beyond Exception: Crashing Android System by Trapping in "uncaughtException"

Jingzheng Wu*†, Shen Liu*, Shouling Ji‡§, Mutian Yang*, Tianyue Luo*, Yanjun Wu*† and Yongji Wang*†
*General Department, Institute of Software, The Chinese Academy of Sciences
†State Key Laboratory of Computer Sciences, Institute of Software
‡College of Computer Science and Technology, Zhejiang University
§School of Electrical and Computer Engineering, Georgia Institute of Technology
Email: jingzheng08@iscas.ac.cn

***Abstract*—Android is characterized as a complicated open source software stack created for a wide array of phones with different form of factors, whose latest release has over one hundred million lines of code. Such code is mainly developed with the Java language, which builds complicated logic and brings implicit information flows among components and the inner framework. By studying the source code of system service interfaces, we discovered an unknown type of code flaw, which is named *uncaughtException* flaw, caused by unwell implemented exceptions that could crash the system and be further vulnerable to system level Denial-of-Service (DoS) attacks. We found that exceptions are used to handle the errors and other exceptional events but sometimes they would kill some critical system services exceptionally. We designed and implemented ExHunter, a new tool for automatic detection of this *uncaughtException* flaw by dynamically reflecting service interfaces, continuously fuzzing parameters and verifying the running logs. On 11 new popular Android phones, ExHunter extracted 1045 system services, reflected 758 suspicious functions, discovered 132 *uncaughtException* flaws which have never been known before and generated 275 system DoS attack exploitations. The results showed that: (1) almost every type of Android phone suffers from this flaw; (2) the flaws are different from phone by phone; and (3) all the vulnerabilities can be exploited by direct/indirect trapping. To mitigate *uncaughtException* flaws, we further developed ExCatcher to re-catch the exceptions. Finally, we informed four leading Android phones manufactures and provided secure improvements in their commercial phones.**

***Keywords*-Exception; Android System Service; Vulnerability; DoS Attack;**

I. Introduction

Android is the most successful smartphone operating system in the world, accounting for 80.7% of the total global smartphone sales in the fourth quarter of 2015 and benefiting all the participants in the ecosystem [10]. With new features and updates of phones brought into Android by developers and manufacturers, Android Open Source Project (AOSP) source code evolves continually and the total number of lines of code has over one million [2]. From the bottom up, the code includes Linux Kernel, Hardware Abstraction Level (HAL), libraries, framework and applications, which is mainly implemented by Java and C languages, builds complicated logic and brings implicit information flows among components and the inner framework.

However, if these complicated information flows are not well thought-out, they could break some fundamental components or services and lead vulnerability exploitations to the Android system [1], [11], [16]. For example, prior research reported a serious Android security flaw called the hanging attribute references (Hares) problem, caused by the conflict in the decentralized, unregulated Android customization process and the complicated interdependencies among different Android applications and components [1]. Another research discovered a general design trait in the concurrency control mechanism of the Android system server that could be vulnerable to DoS attacks, and found four unknown vulnerabilities in critical services (e.g., the ActivityManager and the WindowManager) [11]. They named the problems Android stroke vulnerabilities (ASVs), which would continuously block all other requests for system services, followed by killing the system server and soft-rebooting the Android system. Although lots of research has been conducted to study the Android system and vulnerabilities, with over one hundred million lines of code and evolving versions, the security of source code implementations and information flows in Android is still a big challenge.

Exception beyond Exception. Exception handling is the process of responding to the occurrence of anomalous or exceptional conditions and requiring special processing during computation, and often changes the normal flow of program execution [19]. It is usually implemented by specialized programming languages or by computer hardware mechanisms. Java language (the main programming language of the Android framework) uses exceptions to handle errors and other exceptional events. The class *Exception* and its subclasses are a form of *Throwable* that indicates conditions that a reasonable application might want to catch.

In general, an exception is handled or resolved by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an exception handler. However, Android exception handler is more complicated. If the subroutines of the exception are not

well thought-out, e.g., a thrown exception cannot be caught by any prepared catchers, it would finally trap into the *uncaughtException* function in the *UncaughtHandler* class of the Android system. *uncaughtException* is a high-privilege function that would kill the excepted process directly. Once a system level or critical service is killed exceptionally, an Android system would be crashed which further leads to serious flaws. Such flaws could be exploited as a new type of system level DoS attack vulnerability, which has never been studied before to the best of our knowledge.

Our Findings. By studying the code of system service interfaces, especially the exception blocks, we discovered this unknown type of vulnerabilities caused by *uncaughtException*. Whenever a system service traps into *uncaughtException*, an Android system is crashed and soft-rebooted.

To understand the scope and magnitude of the security hazards introduced by *uncaughtException*, we designed and implemented ExHunter, a new tool for automatic detection of this vulnerability by dynamically reflecting service interfaces, continuously fuzzing parameters and verifying the running logs. By evaluating 11 new popular Android phones, which covers Android versions from 4.0.4 to 6.0, ExHunter obtained 1045 system services, 758 suspicious functions, and finally discovered 132 *uncaughtException* flaws which are new vulnerabilities and generated 275 system DoS attack exploitations. The results showed that: (1) most Android phones distributed by Google, Huawei, Lenovo, Samsung, LG, Motorola, HTC and Nubia suffer from this type of vulnerabilities. Hence, it can almost be believed that the results of other un-evaluated phones are the same; (2) the number and details of the vulnerabilities are different from phone by phone according to the exception implementation of system services; and (3) most of the vulnerabilities can be exploited by directly trapping, while others depend on other services' states.

Enhancements. To mitigate this new type of *uncaughtException* vulnerabilities, we further developed a simple yet effective protection, named ExCatcher, to re-catch the exceptions by filtering the thrown exceptions to ensure that critical system services should not be killed by the high privilege function. ExCatcher maintains a whitelist filled with the discovered *uncaughtException* flaws. Whenever a service traps into the *uncaughtException* exception, ExCatcher immediately checks the whitelist and determines how to deal with the exception. If the trapped service is critical, i.e., killing the system service would lead to an Android crashing, ExCatcher will pass the exception without doing anything to avoid the system rebooting. Otherwise, it executes as usual. Although ExCatcher is a simple mitigation method, which could be further improved by Android manufacturers when customizing their own phones, it is actually easy to be implemented and efficient.

Finally, we reported the discovered vulnerabilities in our research to four leading Android phones manufacturers, and provided secure improvements to their commercial phones.

Contributions. Specifically, we made the following contributions in this paper.

- **New Findings.** Based on the understanding of the Android system and the exceptions, we discovered *uncaughtException* flaws, a new category of Android vulnerability never known before. The problems are caused by exceptionally trapping in *uncaughtException* and trigging process killing by the high-privilege function. We believed most Android phones suffering from this type of vulnerabilities, and most of the vulnerabilities can lead to system level DoS attacks.
- **New Techniques.** We developed ExHunter and ExCatcher to automatically detect *uncaughtException* flaws from the Android phones and protect them against exploitations. With the tools, we finally discovered 132 *uncaughtException* vulnerabilities, generated 275 system DoS attack exploitations and mitigated them by providing re-catching protection techniques.
- **Implementation, Evaluation, and Application.** We implemented ExHunter and ExCatcher, evaluated them with 11 and 3 different Android phones, and the results show that they are effective to detect and mitigate *uncaughtException* flaws respectively. With these implementations, the discovered vulnerabilities and the secure improvements are adopted by four manufacturers in their commercial phones.

II. Background

A. Android System Services

The huge success of the Android system is partly attributed to factors such as the open ecosystem and the feature rich application programming interfaces (APIs) [1], [14], [20]. Furthermore, system services manage almost everything on the Android platform, including about 60-100 services, e.g., WifiManager, BluetoothManager, WindowManager, PackageManager, AudioManager, BackupManager, BatteryManager, ConnectivityManager, etc. They provide the applications with the information and capabilities necessary to work, play key roles in exposing the low-level functions and live from boot to reboot.

In Android applications, services are typically used to perform background operations that take a considerable amount of time. This ensures faster responsiveness to the main thread (e.g. the UI thread) of an application, with which the user is directly interacting. The life cycle of the services used in applications is managed by the Android Framework, i.e., these services have *startService()*, *bindService()* and *stopService()* calls that are called when an activity (or some other components) starts, binds or stops a service.

System services act as the core of the Android system, providing fundamental contexts to various tasks and requests. Whenever the system services fail exceptionally, it would be disastrous to the whole Android system.

B. Exception Mechanisms

Exception is an event that occurs during the execution of a program, switching normal control flows to outside code according to the exceptional condition. An exception that can be caught by a try block is a caught exception, otherwise it is an uncaught exception. The handling code of a caught exception has already been pre-defined. Uncaught exceptions represent instances where a program encountered fatal unexpected conditions at runtime, usually terminate the program and print an error message to the console showing debug and stack trace information.

Uncaught exceptions are often avoided by having a top-level handler that catches exceptions before they reach the runtime environment. In the Android system, the resolution is to create an *UncaughtExceptionHandler* [2]. An *UncaughtExceptionHandler* is an interface defined on the *Thread* class in the Android Software Development Kit (SDK). Creating an instance of *UncaughtExceptionHandler* relies on an object that one wants to handle any uncaught *Throwable* of a thread. Uncaught *Throwable*s are defined as those that are not dealt with (for example by try/catch) and would end up by terminating the thread.

However, we discovered some not well thought-out *UncaughtExceptionHandler* in critical system services which may cause serious problems to the Android system. For example, accessing an array beyond its bounds usually causes a *java.lang.ArrayIndexOutOfBoundsException* to be thrown and kills the main thread and the application. Whenever this main thread belongs to the system service, Android system would be crashed and rebooted.

C. Android Security Problems

Android security has been extensively studied since the emergence of the Android system [6], [14], [17]. The vulnerabilities exist in the whole Android software stack, including Linux kernel, middle-wares, libraries and APIs, the application framework and various applications [7], [11], [15]. However, although the vulnerabilities are noticed and published, researchers found that on average 87.7% of Android phones are still exposed to at least one of 11 known critical vulnerabilities [15], implying that the security situation remains serious.

Similar to the prior security work, we discovered a new unknown type of vulnerabilities caused by not well thought-out *uncaughtException*, which is more sharable than the Stagefright, and has a wider range of influence.

D. Android Security Enhancements

To protect the Android system, enhancements have also been extensively studied [4], [9], [18]. By default, Android uses the concept of *sandbox* to ensure that each application runs with a distinct system identity, and also uses *permission* to allow or deny each application accesses to the phone's resources such as files and directories, network, sensors, and APIs in general [8]. Beside the inner mechanisms, SEAndroid [13] and some other extensions have also been proposed to defend against various vulnerabilities [5], [9].

However, facing with an unknown vulnerability, all of these protections may fail. In this paper, we developed ExCatcher to re-catch the exceptions. Therefore, it can mitigate the *uncaughtException* vulnerabilities and has been adopted by four manufacturers in their commercial phones.

III. UNCAUGHTEXCEPTION VULNERABILITIES

As mentioned earlier, exceptions that are not well thought-out, e.g., a thrown exception cannot be caught by any prepared catcher, would finally trap into the *uncaughtException* function in an Android system. The *UncaughtHandler* class kills the exceptional process straightforwardly regardless of the process' attributes. However, when a system level or critical service is killed exceptionally or un-carefully, the Android system would be crashed and further leads to serious security problems.

A. uncaughtException Exploitations

We discovered this new type of *uncaughtException* vulnerabilities, and systematically analyzed them in this paper, obtaining 1045 system services, 758 suspicious functions, and finally finding 132 exception flaws which are new vulnerabilities. To understand the security risks they may pose, we built end-to-end attacks on the discovered *uncaughtException* vulnerabilities and demonstrated the system DoS attack exploitations. According to the dependence to the phone states, we classified the found *uncaughtException* vulnerabilities into two categories: direct exploitation vulnerabilities and indirect exploitation vulnerabilities.

A surprising finding of our research is that most of the discovered *uncaughtException* vulnerabilities are one stroke vulnerabilities, meaning that a single service invoking will crash the Android system.

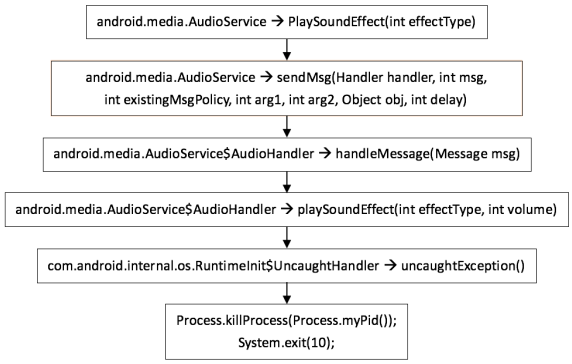

Figure 1. Control Flow Graph for Direct Exploitations of *Uncaughtexception* Vulnerabilities.

Figure 1 contains one sample showing direct exploitations of Android *AudioService uncaughtException* vulnerabilities. To invoke a system service, an attacker could send an intent

whose target function is *PlaySoundEffect* with the integrity parameter of *effectType* larger than a certain integer to Android *android.media.AudioService*. This Android service uses function *sendMsg* to send parameter *effectType* to a callback function *handleMessage* in an inner class *AudioHandler*. Subsequently, function *playSoundEffect* is called and the condition statement *SOUND_EFFECT_FILES_MAP [effectType][1]* is determined. Because the array length is *9*, any value larger than *8* would lead *Out of Bound of Array* problems. However, this exception is not caught in the Android system and thus traps the process into the *UncaughtHandler* class. Finally, *Process.killProcess()* is invoked. *Process.killProcess()* is a high-privilege function that would kill any excepted process directly by its process id (*pid*). In this example, *android.media.AudioService* is killed, and the Android system is crashed.

B. uncaughtException Analysis

Two examples of *uncaughtException* vulnerabilities are demonstrated in the previous subsection. Both of the exploitations send intents to Android system services, exceptionally trap into *uncaughtException* and crash the Android system at last. Essentially, Android system services provided by the system processes have higher privileges to call the corresponding functions, such as operating the underlying drivers. They serve other processes through the Inter Process Communication (IPC) mechanism in the form of cross process Java methods which typically are packaged into the application interface (API) in an Android system. It is generally considered that system service is the basis of Android. If the services are terminated exceptionally, the Android system will be crashed and rebooted.

Taking the direct exploitation in Figure 1 as an example, parameter *effectType* is packaged into an IPC transaction, and assigned to another thread for processing through the Android *Handler* mechanism. The IPC mechanism is synchronized, where any exception in processing will be written directly to the IPC feedback data and returned to the requesting process. However, the introduction of the *Handler* mechanism changed IPC into an asynchronous process. In this case, any exception that is thrown by other processes and not caught will eventually arrive Android Java Runtime Environment, i.e., Dalvik or Android Runtime (ART), which deals with the *uncaughtException* in the *UncaughtHandler* class. According to the implementation of the *killProcess* function as shown in Listing 1, the *uncaughtException* exception thrown by the Dalvik/ART exception handling code will terminate system services and crash the Android system, which is equivalent to a system-level DoS attack.

```
1 private static class UncaughtHandler
    implements
    Thread.UncaughtExceptionHandler {
2   public void uncaughtException(Thread
      t, Throwable e) {
3     Process.killProcess(Process.myPid());
4     System.exit(10);
5   }
6 }
```

Listing 1. Code Snippet of *uncaughtException* in the Android Runtime System.

From the above discussion, the not well thought-out code of exception handling, especially for the system services, may arrive to the Dalvik/ART exception handling code, launch an *uncaughtException* vulnerability and crash the Android system.

C. Challenges

This new discovered vulnerability is mainly because of the not well thought-out exception implementation, which can be exploited by various system services and crashes the Android system. To make matters worse, it is not clear how many and how serious these vulnerabilities are. Therefore, it is necessary to develop tools to detect and mitigate the *uncaughtException* vulnerability for the Android system. To achieve this goal, we have the following challenges.

How to detect the vulnerabilities for each Android system? The number and details of system services are different from phone by phone. More specifically, because of the customization by manufacturers, even the same service may have different interfaces in different phones. Therefore, a detection tool is needed to dynamically obtain the service interfaces and detect vulnerabilities for every phone.

How to mitigate the vulnerability for vulnerable Android systems? When it is clear how many *uncaughtException* vulnerabilities in a certain Android phone, a new protection method is required to mitigate the threats. Furthermore, this method also requires to be easily deployed on Android systems.

IV. ExHunter

To better understand the *uncaughtException* vulnerability of the Android system, we built a new tool, namely ExHunter, an automatic detector that detects *uncaughtException* flaws from Android phones and catches the attempts to exploit *uncaughtException* vulnerabilities on a phone.

A. Design of ExHunter

Due to the large set of source code of the Android system and the deep fragmentation of the Android ecosystem, one can hardly know the number and details of the system services and the *uncaughtException* vulnerabilities on different phones. Therefore, we will obtain the service interfaces dynamically, and refine and verify the candidate *uncaughtException* flaws on the fly.

To achieve this goal, we design ExHunter which could be installed on Android phones as an application. ExHunter consists of five components as shown in Figure 2: ① a lightweight but efficient dynamic listing module to list

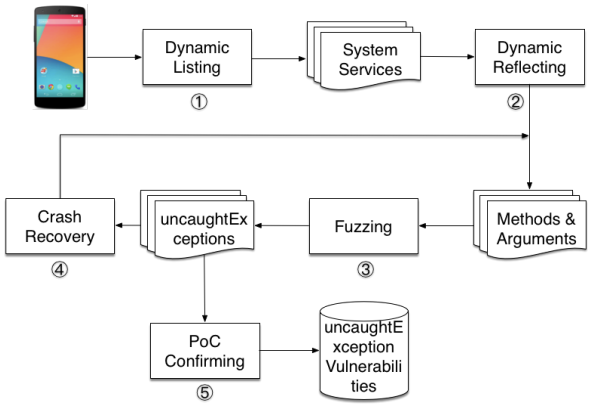

Figure 2. The Design of ExHunter.

system services for an Android phone; ② an inner JAVA reflection module to obtain methods and parameters for each system service interface; ③ a Fuzzing module to mutate the parameters with random values and record the candidate *uncaughtException* flaws; ④ a recovery module to revoke a detection process from the last rebooting method repeatedly; and ⑤ a confirming module uses PoCs to verify whether the candidate *uncaughtException* results are real vulnerabilities. Once a candidate *uncaughtException* flaw is tested and the Android system is crashed, it will be reported and stored in the vulnerability database for further analysis and process.

1) Dynamically Extracting Android Interfaces: For each Android system, we first run a systematic analysis to extract all the system services.

Extracting System Services. Most system services offer fundamental Android features, including display and touch screen support, telephone and network connectivity, which are implemented in Java and some fundamental ones are written in C. For each Android phone to be detected, ExHunter first extracts all the system services with Android inner mechanisms that allow processes to discover and obtain references of system services as needed.

To enable service discovery, the Android *Binder* framework has a single context manager, which maintains references to *Binder* objects. Android's context manager implementation is the *serviceManager* native background process. It is started in the boot process so that system services can register with it as they start up, where services pass service names and a *Binder* reference to the service manager. After service registration, any client can obtain its *Binder* reference by using its name. However, most system services implement additional permission checks. Hence, obtaining a reference does not automatically guarantee a complete service discovery.

In our design, ExHunter uses the service list command to obtain a list of registered services, which returns the name of each registered service and the implemented *IBinder* interface. After reading from the *bufferedReader*, all Android system services are obtained.

Reflecting System Service Interfaces and Attributes. With a few exceptions, each system service defines a remote interface that can be called from other services and applications. The service interfaces are exposed directly in the framework and could be accessed via facade classes called managers. For example, the system service that controls the Wi-Fi connectivity is *WifiService*, which offers a public interface via the *WifiManager* facade class. The *WifiService* delegates the Wi-Fi state management to a rather complex *WifiStateMachine* class, which can go through more than a dozen states while connecting to a wireless network.

To obtain the interfaces which would be tested in the following steps, ExHunter uses the Java reflection mechanism module to get methods and parameters for each system service. Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. It is a relatively advanced feature and a powerful technique that can enable applications to perform operations which would otherwise be impossible. Finally, after reflecting each of the system services, all the methods and the corresponding attributes (parameters and types) are obtained.

2) Dynamically Trapping Android Interfaces: After extracting Android system services and the corresponding interfaces, we build test cases to trap into *uncaughtException* exceptions. Here, ExHunter uses fuzzing techniques and catches the running states of the system services. When an Android system crash is caught, a new *uncaughtException* vulnerability is found.

Fuzzing and Catching Exceptions. Each system service has exposed interfaces for being called from other services and applications. Random or dangerous calls to interfaces may trap system services into *uncaughtException* exceptions, where fuzzing might be the best technique to achieve this goal. Fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. The key step of fuzzing is to define lists of "known-to-be-dangerous values" (fuzz vectors) for each type, and to inject them or recombinations.

In our design, ExHunter randomly generates parameter values, e.g., integers (zero, possibly negative or very big numbers), chars (escaped, interpretable characters or instructions), strings (null, spaces or long strings), etc., for each interface of a system service. After dynamic generation, ExHunter packages the values into a *Parcel* data structure and sends it through the *Binder* mechanism.

A meta *parameterTypes* array is initialized, with which ExHunter mutates and generates the values by the fuzzing technique. Then, ExHunter sends the parameter values to an interface continually, and monitors the results. If a service traps into the *uncaughtException* exception, i.e., the Android

system is crashed, ExHunter will report it and wait for verification. ExHunter repeats this process for each service, and sends the detection results to the following steps.

Recovering *uncaughtException*. When a system service traps into *uncaughtException* exception, the Android system would crash, meaning the phone would reboot. In general, applications would not start until receive a notification, which could be a touch activity or a broadcast. Broadcasts can originate from the system (e.g., announcing changes of the network connectivity) or from a user application (e.g., announcing that the background data update has completed). Any application registered as a broadcast receiver will receive the event and respond to the systemwide event.

To monitor the reboot event, ExHunter registers as a *BroadcastReceiver*, listens to the *BOOT_COMPLETED* action and reacts to it. *BOOT_COMPLETED* is a Broadcast Action that is broadcasted once after the system has finished booting. ExHunter overrides the *onReceive* function, where ExHunter saves the last crashing *uncaughtException* context and restarts the fuzzing process sequentially.

Verifying *uncaughtException* Vulnerabilities. As described before, any exception thrown by a system service that is not caught will finally trap into the *uncaughtException* exception, which is the exception handling mechanism of Android Dalvik/ART. When Android is crashed, the *uncaughtException* running tracks are recorded and a proof of concept (PoC) program is obtained.

```
1 E/AndroidRuntime(12800): *** FATAL
    EXCEPTION IN SYSTEM PROCESS:
    WifiStateMachine
2 E/AndroidRuntime(12800):
    java.lang.NullPointerException
3 E/AndroidRuntime(12800): at
    android.net.wifi.WifiStateMachine
    $DriverStartedState.processMessage(
    WifiStateMachine.java:2826)
```

Listing 2. Log Snippet of *uncaughtException* Vulnerability Error Message.

As the code snippet shown in Listing 2, a runtime error message of the *uncaughtException* vulnerability is recorded. In this example, the message shows that the source of the fatal exception is *WifiStateMachine*, caused by the *NullPointerException* exception in Android *WifiService*. With this error message, an *uncaughtException* vulnerability that can crash the Android system can be reproduced easily.

B. Implementation of ExHunter

We implemented the approach presented in the previous subsection as an Android application (APK), named ExHunter, which is aiming to automatically detect the *uncaughtException* vulnerability by dynamically reflecting service interfaces, fuzzing parameters and verifying the running logs. To detect the *uncaughtException* vulnerabilities on different Android phones, ExHunter needs to run on most compatible phones, e.g., Android versions from 4.0.4 to 6.0. We granted ExHunter enough permissions in its *manifest*, with which it could access the appropriate resources at runtime. In the following subsection, we will describe how ExHunter is used in *uncaughtException* vulnerability detection.

C. Evaluation of ExHunter

Using ExHunter, we performed a measurement study to inspect 11 popular Android phones, found 132 *uncaughtException* flaws which are new vulnerabilities, demonstrated the PoCs and reported them to four manufacturers.

1) uncaughtException Results and Analysis: In our evaluation, we collected 11 Android phones from Google, Huawei, Lenovo, Samsung, LG, Motorola, HTC and Nubia. The phones are customized by the vendors and with different versions, i.e., the Android versions are from 4.0.4 (whose code name is Ice Cream Sandwich) to 6.0 (whose code name is Marshmallow). The detailed information is presented in Table I, including the vendors, the models, the versions, the number of system services, the reflected interfaces, the discovered vulnerabilities and the attack PoCs.

Taking Google Nexus 6P as an example, it belongs to the Google Nexus series, which is released in Oct 2015 by Google as the flagship Android phone to demonstrate Android's latest software and hardware features. It is generally believed that the Google Nexus series are more secure than the phones released by other manufacturers. As we can see from Table I, for Nexus 6P which is Android 6.0 Marshmallow, ExHunter extracted 110 system services, reflected and fuzzed 80 interfaces, detected 15 *uncaughtException* flaws and obtained 39 DoS attack PoCs. This detection process takes 72 hours. Nexus 6P has the latest Android version, while HTC T528w has the oldest version of Android 4.0.4. In the evaluation of T528w, ExHunter extracted 74 system services, reflected and fuzzed 55 interfaces, detected 8 *uncaughtException* flaws and obtained 8 DoS attack PoCs. Other phones evaluated in Table I also suffer from the *uncaughtException* vulnerabilities, and the system services, interfaces, vulnerabilities and PoCs are different from phone by phone. These differences are mainly caused by the Android versions and the customization. For example, manufacturers might customize the system services and interfaces according to their own phone market purpose.

We have also evaluated some other phones, and found *uncaughtException* vulnerabilities in all of them. Therefore, according to the evaluations, we believe that the *uncaughtException* vulnerability might be universal in the whole Android world while the details are different depending on the system services.

Numbers of *uncaughtException* Vulnerabilities. From Table I, we can see that Huawei H60-L01 has the most vulnerabilities. However, the services and the interfaces in

Table I
uncaughtException VULNERABILITIES DETECTED BY EXHUNTER FROM 11 ANDROID PHONES.

Vendor	Model	Android Version	Services	Interfaces	Vulnerabilities	PoCs
Lenovo	S90-u	4.4.4	91	69	14	14
Nubia	Nx430A	4.2.2	80	54	10	10
Huawei	H60-L01	4.4.2	93	65	21	21
Samsung	SM-N9008V	4.3	117	84	12	12
Samsung	GT-I9508	4.2.2	106	78	9	9
Google	Nexus 6P	6.0	110	80	15	39
Moto	Moto X	5.0.2	106	81	13	45
LG	G3	4.4.2	116	81	11	98
HTC	T528w	4.0.4	74	55	8	8
Google	Nexus 4	4.2.2	73	52	10	10
Google	Nexus 5	4.4.4	79	59	9	9

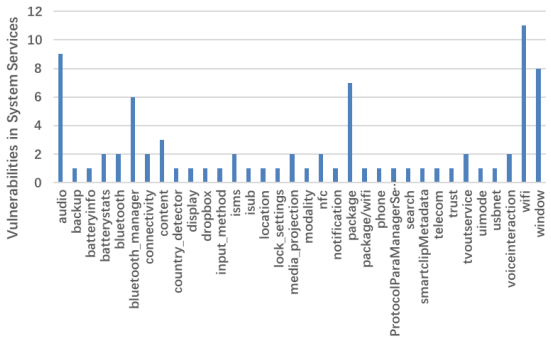

Figure 3. Distribution of *uncaughtException* Vulnerabilities in System Services.

H60-L01 are 93 and 65 respectively, which are not the most among the phones. On the other side, HTC T528w has the least vulnerabilities, while the services and the interfaces are not the least. The number of the *uncaughtException* vulnerabilities is different in the collected phones, but it has no explicit correlation to the Android versions and manufacturers.

From the analysis, we can see that the discovered *uncaughtException* vulnerability is merely related to the not well thought-out exceptions, i.e., whenever a thrown exception cannot be caught by any prepared catcher, it would finally trap into the *uncaughtException* exception and lead to a system level DoS attack.

Distribution of System Services. When a system level or critical service traps into the *uncaughtException* exceptionally, the Android system is crashed and rebooted. By analyzing the collected 11 Android phones, we can see that the *uncaughtException* thrown by 34 system services as shown in Figure 3 can be exploited as vulnerabilities. From Figure 3, the vulnerabilities caused by the Wi-Fi service appear in all the 11 phones, which implies that the Wi-Fi service is more vulnerable than the other services and might be the most easily exploitable attack surface. It obviously concludes that when an attack program is exploited with the Wi-Fi *uncaughtException* vulnerability, most Android phones can be attacked. Other serious services are *audio*, *window*, *package*, *bluetooth_manager*, etc.

From the distribution of the *uncaughtException* vulnerabilities, it is believed that these system services can be exploited easily. Therefore, for manufacturers, when developing and customizing new Android phones, they should pay much more attention to these 34 services, especially Wi-Fi, *audio*, *window*, *package*, and *bluetooth_manager* services.

2) uncaughtException Attacking Analysis: From Table I, we can see that the number of PoCs are either equal or more than the detected vulnerabilities. For example, we found 15 vulnerabilities in Google Nexus 6P, while 39 PoCs are obtained. The reason for this difference is that an interface could be assigned different values, which are generated by the fuzzing process of ExHunter. As shown in Table II, a vulnerable method *setWifiApConfiguration* which belongs to the Wi-Fi system service in Nexus 6P can be exploited by 8 PoCs with the *CHANGE_WIFI_STATE* permission. The differences of these PoCs are the parameter types, i.e., any value meets the type in Table II may lead to an *uncaughtException* attack.

For a typical direct exploitation, ExHunter sends the obtained parameter values and monitors the result. In this example, system service *WifiService* traps into *uncaughtException*. Then, it is killed by *Process.killProcess()*. Finally, the Android system is crashed and rebooted, which proves a direct *uncaughtException* vulnerability.

3) Summarization: Our implementation of ExHunter was found to be effective at detecting *uncaughtException* and generating PoCs. We evaluated ExHunter with 11 popular Android phones. It extracted 1045 system services, reflected 758 suspicious functions, discovered 132 *uncaughtException* new vulnerabilities and generated 275 PoCs used for system DoS attack exploitation. The results show that: (1) most Android phones distributed by Google, Huawei, Lenovo, Samsung, LG, Motorola, HTC, and Nubia suffer from this type of vulnerabilities. Thus, it can be believed that the conclusion can be extended to other un-evaluated phones; (2) the number and details of the vulnerabilities are different from phone by phone; and (3) most of the vulnerabilities can be exploited by directly trapping, while others depend on some services' states. We reported all the detected *uncaugh-*

Table II
TYPICAL METHOD AND PARAMETERS IN AN *uncaughtException* VULNERABILITY.

System Service Method	Parameter Type	Parameter
setWifiApConfiguration (android.permission.CHANGE_WIFI_STATE)	Map	Map.put(789)
	Array	new String[]{"1"}
	BinderArray	new IBinder[]{ib}
	BooleanArray	new boolean[]{false}
	DoubleArray	new double[]{1, 2, 3, 4}
	SparseBooleanArray	SparseBooleanArray.append(0, false)
	StringArray	new String[]{"123"}
	StrongBinder	getIBinder(sername)

tException vulnerabilities and exploitations to four phones manufacturers, who confirmed our findings and acknowledged our contributions. To mitigate the *uncaughtException* vulnerability, we propose ExCatcher to catch the exceptions in the following section.

V. EXCATCHER

Based on the cause of the general *uncaughtException* flaws, we designed a protection extension, named ExCatcher, an Android patch that re-catches the *uncaughtException* exception and avoids the critical system services to be killed exceptionally.

A. Design of ExCatcher

Fundamentally, the cause of *uncaughtException* flaws is that the uncaught exceptions are thrown to the runtime exception mechanism in the Android framework, which would kill the process directly with high privilege. Whenever a critical system service is killed, the Android system is crashed and rebooted. Therefore, the *uncaughtException* flaws can be fixed by phone manufacturers, who are supposed to either put proper security checks in each system service interface or rewrite the *uncaughtException* exceptions in their code.

In this paper, we developed a simple yet effective protection method to re-catch the *uncaughtException* exceptions in the Android framework code, called ExCatcher. ExCatcher filters the thrown exceptions to ensure that critical system services will not be killed by the high privilege function. ExCatcher collects the discovered *uncaughtException* flaws and adds them to a whitelist. Whenever a service traps into the *uncaughtException* exception, ExCatcher immediately checks the whitelist before the service being killed. If the service is critical, e.g., a system service that would lead to Android crash, ExCatcher will pass the exception without doing anything to avoid the system rebooting. If the system service is not in the whitelist, there is no difference than before. Although ExCatcher is a simple mitigation method, which could be further improved, it is actually easy to be implemented and it is efficient in practice.

B. Implementation of ExCatcher

We implemented ExCatcher using a source code patch that can be patched to the *RuntimeInit.java* file in the Android framework. *RuntimeInit.java* initializes the application runtime environment for an Android system, where we mainly focus on the *uncaughtException* function in the *UncaughtHandle* class. *uncaughtException* logs a message when a thread exits due to an uncaught exception, catches the exception for the main thread, and kills the process using *Process.killProcess(Process.myPid())*.

For a phone manufacturer who has the source code of the Android system, ExCatcher could be implemented easily by applying this source code patch. Furthermore, if new *uncaughtException* flaws discovered in the future, the only thing to do is to add the flaws to the whitelist. However, the best way we recommended to manufacturers is to build a plug-in library to maintain the whitelist dynamically.

C. Evaluation of ExCatcher

We implemented ExCatcher and evaluated it with Google Nexus series phones (Nexus 4, Nexus 5 and Nexus 6P), which can be built from the AOSP source code. The source code of ExCatcher is patched to a local Google AOSP mirror code repository. Then, we built the improved source code and obtained new Android image. Taking Google Nexus 6P as an example, after flashing the new image, we re-run the discovered 39 *uncaughtException* DoS attack exploitations, and none of them can crash the Android system again. In our evaluation, ExCatcher passes the *uncaughtException* exceptions without any further actions, which induces no side effects. This is mainly because the vulnerable interfaces are the configuration actions, even the actions failed and the requiring services ignored, there have little effects.

We also evaluated Nexus 4 and Nexus 5 phones, and the results are the same with Nexus 6P. Therefore, we provided ExCatcher to some manufacturers, and after deploying, they confirmed the method and improved their phones according to it. However, according to the agreements signed with them, we do not list them here.

VI. DISCUSSION

The *uncaughtException* vulnerability is an implementation lapse in the Android exception mechanism. Android is a complex system, and a slight lapse may lead to serious problems, e.g., a Wi-Fi setting action with a certain parameter traps into the *uncaughtException* exception and crashes the

Android system. As far as we known, this new vulnerability is first discovered in this paper.

Detection of *uncaughtException* Vulnerabilities. ExHunter is presented to automatically detect *uncaughtException* flaws from Android phones. Although ExHunter is effective and has discovered 132 *uncaughtException* new vulnerabilities and 275 PoCs that used for system DoS attacks, it could be further improved in the following ways.

As presented before, ExHunter used the fuzzing technique and found 34 vulnerable system services on 11 Android phones, while more vulnerable services could be found if a more optimized fuzzing algorithm is developed and leveraged, e.g., fuzzing with machine learning or deep learning.

Firstly, more optimized fuzzing algorithm could be introduced to ExHunter, e.g., machine learning and deep learning techniques. Therefore, we will improve the fuzzing technique of ExHunter in the future. Secondly, due to the complicated state dependency, ExHunter may not discover all the indirect. Therefore, we will study the relation between the indirect *uncaughtException* vulnerabilities and the states of an Android system.

Mitigation of *uncaughtException* Vulnerabilities. ExCatcher is presented to re-catch the *uncaughtException* exceptions, with which the manufacturers can fix *uncaughtException* flaws easily. We have implemented ExCatcher in Google Nexus 4/5/6P, and did not find side effects until now. However, to determine whether there are side effects brought by ExCatcher, it needs a long running time.

In this paper, the using of ExCatcher requires rebuilding the Android system and re-flashing the generated image. For the already distributed Android phones, this is difficult. Therefore, we will study how to implement ExCatcher as a plug-in library or a stand-alone Android application, which could be maintained dynamically on the phone.

VII. Related Work

Vulnerability is the main obstacle for Android market expansion, since billions of Android phones may be vulnerable to the discovered vulnerabilities [3], [16], [21]. For instance, Stagefright is a collection of bugs, discovered by Joshua Drake on July 27, 2015 [12], allowing the attackers to remotely execute malicious code on more than one billion Android phones. To make matters worse, the vulnerabilities appear almost everyday. In this section, we review related research and compare our work with those studies.

Android stroke vulnerability (ASV). Prior research [11] discovered a general design trait, named ASV, in the concurrency control mechanism of the core of the Android system server that could be vulnerable to DoS attacks. They found that once any one of the monitor locks cannot be acquired in a preset period under the Android concurrency control mechanism, the watchdog considers that the corresponding system service is in the starving/deadlock situation. Then instead of killing the relevant service thread, it kills the whole system server and forces the Android user-space (e.g., the zygote, the system server and other processes) to be rebooted. Therefore, when an application can potentially control the watchdog to interrupt the whole system server process (e.g., the application invoke some specific action to raise a false alert to the watchdog thread), it will launch a system level DoS attack.

The *uncaughtException* vulnerability is similar to ASV, e.g., it also can launch a DoS attack to an Android system in exploitation. However, there are some differences between them. Fundamentally, the cause of the *uncaughtException* vulnerability is that the uncaught exceptions are thrown to the runtime exception mechanism in the Android framework, which can kill the exception process directly with high privilege and lead an Android system to crash. The *uncaughtException* vulnerability has no relation to the concurrency control mechanism, and is another new vulnerability.

Hanging attribute references (Hares) vulnerability. Another prior research [1] reported a serious Android security flaw called the Hares problem, caused by the intrinsic interdependent relations between different Android components. Generally, the components (applications and framework services) connect one party to another through references to the latter's attributes such as package, activities, services names, authorities of content providers and permissions, e.g., startActivity called by one application to invoke another's activity (whose name is specified through *setClassName*). However, the not well thought-out components can easily break some of such relations, resulting in the references to non-existing attributes (e.g., the authorities of the SMS/MMS providers not on the tablet), which leads to *Hares* vulnerabilities.

The *uncaughtException* vulnerability is similar to Hares, e.g., they are essentially caused by the not well implementation in the Android system. However, the difference is that the *uncaughtException* vulnerability is caused by the not well thought-out runtime exception mechanism, while *Hares* is caused by the not well thought-out components in customization.

Android Security Mechanisms To defend against the discovered vulnerabilities, researchers have proposed many defense mechanisms. For example, [11] reported all the identified ASVs to Google's security team, who confirmed their findings and acknowledged their contributions. CVE IDs for the ASVs will be generated after fully patching. Based on the understanding of the cause of this general design flaw, they proposed some mitigations and defenses, including user side remediations, access control mechanisms, resource usage thresholds, concurrency control improvement and smart watchdog mechanism redesigning. Prior research [1] developed a simple protection, using an app, called HareGuard, to scan other third-party applications whenever it is installed to ensure that it is not taking advantage of any known Hare vulnerabilities on a specific phone model.

Some other security enhancements are also proposed [5], [9], [13]. For example, Security Enhanced Android (SEAndroid) released by The National Security Agency in 2012 is a flexible mandatory access control to Android [13].

Similar to the prior research, we design a simple yet effective protection ExCatcher to mitigate the *uncaughtException* vulnerabilities by re-catching the *uncaughtException* exceptions in the Android framework code. We provided our findings to four manufacturers, and recommended them to re-implement ExCatcher as a plug-in library or a stand-alone Android application in their phones.

VIII. Conclusions

In this paper, we reported our research on a serious Android security flaw that has not been studied before. The problem, called the *uncaughtException* vulnerability, is caused by the uncaught exceptions thrown to the runtime exception mechanism in the Android framework, which would kill a process directly with high privilege. Whenever a critical system service is killed, the Android system is crashed and rebooted. We further built the first tool for automatically detecting *uncaughtException* vulnerabilities and leveraged it to analyze 11 popular Android phones, and discovered 132 *uncaughtException* new vulnerabilities and built 275 PoCs used for system DoS attack exploiting. To mitigate the *uncaughtException* flaws, we further developed ExCatcher to re-catch the exceptions and mitigate the vulnerabilities. ExCatcher can be extended and re-implemented as a plug-in library or a stand-alone Android application. Finally, our research has been reported to four manufacturers and helped them improve their new commercial phones.

Acknowledgments

The authors are grateful to the anonymous reviewers for their time and valuable comments. This work is supported by the research funds under No. 61303057, No. 2012ZX01039-004, No. 2016C01G2010916 and No. CCF-Tecent AGR20160109.

References

[1] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *CCS'15*, pages 1248–1259.

[2] Android. Welcome to the android open source project! http://source.android.com.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *NDSS'14*.

[4] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *SP'15*, pages 931–948.

[5] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *SPSM'11*, pages 51–62.

[6] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *SEC'15*, pages 659–674.

[7] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *SEC'11*, pages 21–37.

[8] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *S&P*, 7(1):50–57.

[9] W. Enck, M. Ongtang, and P. D. McDaniel. On lightweight mobile phone application certification. In *CCS'09*, pages 235–245.

[10] Gartner. Gartner says worldwide smartphone sales grew 9.7 percent in fourth quarter of 2015. http://www.gartner.com/newsroom/id/3215217.

[11] 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 *CCS'15*, pages 1236–1247.

[12] J. jduck Drake. Stagefright: Scary code in the heart of android. https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf.

[13] S. Smalley and R. Craig. Security enhanced (SE) android: Bringing flexible MAC to android. In *NDSS'13*.

[14] Sufatrio, D. J. J. Tan, T. Chua, and V. L. L. Thing. Securing android: A survey, taxonomy, and challenges. *ACM Comput. Surv.*, 47(4):58, 2015.

[15] D. R. Thomas, A. R. Beresford, and A. Rice. Security metrics for the android ecosystem. SPSM '15, pages 87–98.

[16] H. Zhang, D. She, and Z. Qian. Android root and its providers: A double-edged sword. In *CCS'15*, pages 1093–1104.

[17] M. Zhang, Y. Duan, Q. Feng, and H. Yin. Towards automatic generation of security-centric descriptions for android apps. In *CCS'15*, pages 518–529.

[18] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *SP'15*, pages 915–930.

[19] P. Zhang and S. G. Elbaum. Amplifying tests to validate exception handling code. In *ICSE'12*, pages 595–605.

[20] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *SP'14*, pages 409–423.

[21] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS'12*.