NativeX: Native Executioner Freezes Android

Qinsheng Hou
QI-ANXIN Technology Research Institute
Legendsec Information Technology (Beijing) Inc.
Beijing, China
houqinsheng@qianxin.com

Yao Cheng
Huawei International
Singapore, Singapore
chengyao101@huawei.com

Lingyun Ying*
QI-ANXIN Technology Research Institute
University of Chinese Academy of Sciences
Beijing, China
yinglingyun@qianxin.com

ABSTRACT

Android is a Linux-based multi-thread open-source operating system that dominates 85% of the worldwide smartphone market share. Though Android has its established management for its framework layer processes, we discovered for the first time that the weak management of native processes is posing tangible threats to Android systems from version 4.2 to 9.0. As a consequence, any third-party application without any permission can freeze the system or force the system to go through a reboot by starving or significantly delaying the critical system services using Android commands in its native processes. We design *NativeX* to systematically analyze the Android source code to identify the risky Android commands. For each identified risky command, NativeX can automatically generate the PoC (Proof-of-Concept) application, and verify the effectiveness of the generated PoC. We conduct manual vulnerability analysis to reveal two root causes beyond the superficial attack consequences. We further carry out quantitative experiments to demonstrate the attack consequences, including the device temperature surge, the battery degeneration, and the computing performance decrease, based on which, three representative PoC attacks are engineered. Finally, we discuss possible defense approaches to improve the management of Android native processes.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; **Denial-of-service attacks**.

KEYWORDS

Android security, mobile system security, denial-of-service attacks, Android native processes, Android commands

ACM Reference Format:
Qinsheng Hou, Yao Cheng, and Lingyun Ying. 2020. NativeX: Native Executioner Freezes Android. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3320269.3384713

*Corresponding author.

ASIA CCS '20, October 5–9, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/10…$15.00
https://doi.org/10.1145/3320269.3384713

1 INTRODUCTION

Android is a Linux-based multi-thread open-source operating system for mobile devices. It not only has gained tremendous popularity among mobile users in recent years, but also has been used in IoT devices and various mission-critical tasks, such as point-of-sale devices [11], medical devices [30], on-vehicle systems [5][6], and even aircraft and satellite devices [1][10]. One of the reasons that Android has been widely adopted is that Android provides compatible development support for programming in both framework layer and native layer, so that developers can freely enjoy both the transparent access to Android resources in the framework layer and the low running latency and high protection in the native layer.

The fundamental resources, i.e., system resources and device computing resources, are shared across the Android system for both framework layer processes and native layer processes. In order to avoid conflict during the use of the system resources, the access to such resources is coordinated by the synchronization mechanism. The synchronization mechanism works by introducing an exclusive lock that only one thread at a time can acquire the lock (except the read lock) and the lock is required to be acquired prior to any access to the shared resource [7]. However, it also implies that if a process has occupied a system resource, the other processes waiting for the resource are blocked until the resource becomes free. Similarly, if a process has taken a large portion of the device computing resources, e.g., CPU or RAM, the other processes may run slow due to the lack of computing resources.

In case such an undesirable situation happens, Android provides a set of process management mechanisms for different situations to ensure the proper functioning of the system. For the situations where an app is not responding, the device memory is low, and a system service is not responding, Android triggers Application Not Responding (ANR), Low Memory Killer (LMK), and the Android watchdog mechanism, respectively. The proper functioning of Android relies on the assumption that there is no process that cannot be properly managed (e.g., killing or restarting the process) by any above memory management mechanism.

Unfortunately, this assumption does not always hold. Our work uncovers that an Android native process is such a process that can exhaustively monopolize the system resources or the device computing resources in an unconfined manner. The native process here aligns with the native process concept used in LMK. It refers to Linux processes running in the Android system. A native process

can be from i) the Linux executables shipped in Android system, no matter it can be invoked by third-party apps or not, and ii) the third-party apps through Java API `Runtime.getRuntime().exec()`, native API `fork()` or `system()`. A native process cannot be properly managed by any existing memory management mechanism. Due to the insufficient management of native processes, an app without any permission can result in an unresponsive Android within 5 seconds using the legitimate Android commands as attack vectors.

To the best of our knowledge, this is the first exploration of exploiting Android commands as attack vectors. With the improvement of discreet security mechanisms in Android, it is more and more difficult to exploit APIs as attack vectors. The attacks relying APIs as attack vectors on previous Android versions cannot be reproduced in later versions due to the unavailability of the attack APIs or new constraints in accessing directories or files [22][23][31]. Using legitimate commands as attack vectors is rarely studied which is, in fact, a valuable direction that is worth exploring.

We design and implement an automatic analyzing tool named *NativeX* (Native eXecutioner) to identify the risky Android commands, generate the PoC (Proof-of-Concept) exploit apps, and verify the effectiveness of the generated PoC apps. Note that the Android commands here include both Linux commands (e.g., `top`) and Android-specific commands (e.g., `am`). Specifically, NativeX takes Android source code including both framework source code and command source code as input. It is able to find out the system resources, specifically the storage resources, i.e., files and directories, that are shared by both the framework critical system services and the Android commands. NativeX focuses on storage resources since they have been considered as a type of informative and competing resources [17]. NativeX further automatically constructs a PoC exploit app for each Android command identified in the previous step, and verifies the effectiveness of the app. Our findings show that this type of vulnerabilities existing in a wide range of Android versions from 4.2 to 9.0 which cover 99.7% of more than 2 billion real-world Android devices including smartphones and IoT devices [3][9].

We further conduct manual vulnerability analysis and reveal two root causes. Firstly, there is no restriction on the number of native processes that an Android app can spawn. Secondly, for the native processes that have already started, there is no effective management, that is, Android cannot effectively manage the resources used by native processes even when the system itself is under high resource pressure. The vulnerability analysis shows that, although the attack consequences seem alike, the causes of the identified vulnerabilities are totally different from the previous DoS attacks [13][22].

We quantitatively evaluate the attack consequences, including device temperature surge, battery degeneration, and computing performance decrease under persistent attack. Though the attack consequences may be speculative, they are rarely studied in a quantitative manner. Based on the easily-exploitable nature of the vulnerability, we also design and present three representative PoC attacks, i.e., DoS attack against Android system, DoS attack against Android app, and physical harm to users.

This work suggests that the weak management of Android native processes urgently requires amendment. We also discuss possible defense approaches against this type of vulnerabilities. In summary, this paper makes the following main contributions.

- We reveal a new type of vulnerability existing in a wide range of Android versions from 4.2 to 9.0 that can lead to a system freeze/reboot within 5 seconds, ascribing to the insufficient management of Android native processes. An app without any permission can monopolize a shared resource using legitimate Android commands in native processes to force critical system services to be starved or slowed down, which further freezes the system.
- In order to find out the attack vectors exploiting such vulnerability, we design and implement an automatic static analyzing tool NativeX which is the first exploration of using Android commands as attack vectors. NativeX successfully identifies a list of commands on the 6 major Android versions from 4.2 to 9.0 which can be practically used to launch attacks. It can further construct a PoC exploit app for each identified command and verify the effectiveness of the PoC app.
- We conduct experiments to qualitatively demonstrate the attack consequences of the vulnerability in three aspects, i.e., the temperature surge, the battery degeneration, and the computing performance decrease. We also showcase three representative attacks including the DoS attack against Android system, the DoS attacks against target Android application, and the physical harm to device users due to the device overheat.
- We conduct vulnerability analysis and reveal the root causes of the vulnerability, based on which, we discuss the defense approaches against such vulnerability and provide our mitigation analysis to the Android security team for their reference.

2 BACKGROUND AND VULNERABILITY OVERVIEW

2.1 Android Process Management

There are four mechanisms that manage the running processes in Android, i.e., ANR, LMK, Android watchdog, and JVM Garbage Collector.

ANR. ANR is a situation where an application is not responding to an input event or processing a broadcast for a certain amount of time. The ANR mechanism is mainly to guarantee the smooth performance of Android from the user experience angle. ANR is managed according to the running information, e.g., the CPU states and the timeout message, from *ActivityManagerService* (AMS). ANR sets time thresholds for completing different types of operations. If any operation exceeds the preset threshold regardless of the cause of the delay, Android treats the situation as ANR and pops up a dialog providing the user with options either to wait or force quit the unresponsive app. Although the force quit may cause data loss or interim operation interruption, it can generally improve the user experience and reduce the risk of extending the local delay to the system-wide.

LMK. LMK manages the processes from the system memory usage angle. LMK is designed to monitor the memory state of the Android system and react to high memory pressure by killing the least essential processes at that moment to keep the

system performing at acceptable levels. In order to determine the importance of processes, LMK maintains an importance hierarchy based on the `oom_adj` value of the process which has 16 importance values and is calculated based on a list of factors regularly in `ActivityManagerService`. The smaller the process's `oom_adj` value is, the more important the process is. The "native process" is one of the priority levels. The absolute `oom_adj` value for the native process may vary in different Android versions (e.g., -1000 in Android 9.0), but it is the lowest among all the processes within the same Android version and commented as "not being managed" in the source code.

Android Watchdog. Android watchdog is to deal with the system not responding situation. Android watchdog is a separate thread in `system_server` that monitors the most important system services in `system_server`. It is worth noting here that the Android watchdog is different from the traditional Linux watchdog. The Linux watchdog monitors the running status of the whole system, while Android watchdog only monitors the critical system services. These services, such as `ActivityManagerService` which is fundamental in the previous two process management mechanisms, are so critical to the Android system that any failure of these services may lead to the system unavailability. Android watchdog is a time-based monitor that any operation exceeding the preset timer triggers a recovery process. Specifically, Android watchdog has two checkers, i.e., `Monitor Checker` and `Looper Checker`. The monitored services register themselves to watchdog's `Monitor Checker` or `Looper Checker` by adding `Watchdog.getInstance().addMonitor()` or `Watchdog.getInstance().addThread()` to the source code in their service classes. These two checkers obtain the service running status through some operational tests. If there is a timeout occurs, Android watchdog generates necessary log information and sends signal 9 to `system_server` process to kill and restart it. We can see that the Android watchdog is designed to recover the system service from deadlock, starvation and other failures by killing the `system_server` and forcing a system soft-reboot. In this sense, the watchdog is the last line of defense in Android.

JVM Garbage Collector. Other than the three management mechanisms designed specifically for Android, there is one more memory management mechanism inherited from the JVM (Java Virtual Machine). In order to avoid the instability and unresponsiveness in JVM applications caused by memory shortage, JVM Garbage Collector (GC) is to free unreferenced space in the JVM heap memory. GC monitors all the objects in the heap memory and deletes objects that are not referenced by any part of the running application. For Android processes, the framework processes run in JVM and the native processes spawned from the Java code (e.g., `Runtime.getRuntime().exec()`) are also managed by such mechanisms. However, the native processes spawned from the native code through JNI framework use memory in the native heap instead of memory in JVM heap, and hence are not managed by any mechanisms from JVM.

The above four existing process management mechanisms are all focusing on the framework processes. Terminating the app process, e.g., by ANR or LMK, indeed kills all processes in its process group including both framework processes and native processes. However, we have not observed any management mechanism specifically for the native processes, which implies a risk to the Android system.

2.2 Vulnerability Overview

We reveal a new type of vulnerability that is due to the weak management of native processes. The vulnerability can be understood from two angles, i.e., the system resources and device computing resources.

Freezing the system via occupying critical system resources using legitimate Android commands: Android inherits part of the Linux commands, e.g., `top`, meanwhile, it also develops a set of specific commands, e.g., `am`. With the help of Android NDK, any Android app is able to invoke such commands in its native processes. Due to the weak management of the native processes, the Android commands running in the native process can occupy the system resources without any restriction. If the occupied system resource is indispensable to the critical system services, it will lead to the starvation of the critical system services. Such starvation in critical system services freezes the whole system and further triggers the watchdog recovering process that kills the Android userspace (e.g., the system server, the Zygote and other processes) and forces the system to go through a soft-reboot.

Exhausting the device computing resources via spawning a large number of native processes: As there is no restriction on the use of the native process, any app is able to spawn a large number of native processes in an instant to exhaust the device computing resources. Unfortunately, due to the lack of computing resources, ANR and LMK cannot timely react to this situation to kill the malicious app. As a consequence, the system is not able to maintain its basic functionality because of the unavailability of the device computing resources. Finally, it falls again to the last defense line, i.e., the watchdog recovering process.

3 NATIVEX

3.1 Overview

As illustrated in Fig. 1, NativeX is composed of four modules represented in dashed rounded rectangles, i.e., static code analyzer, risky command analyzer, automatic PoC generator, and automatic verifier. The static code analyzer takes the Android framework source as input to sift out a set of critical file system resources that are used in the potential risky Android framework methods. Risky command analyzer further identifies the Android commands that have operations on these critical file system resources by analyzing the Android command source code. The static code analyzer and the risky command analyzer together identify the Android commands that can potentially be abused to occupy the system resources, particularly the file system resources that are required by critical Android system services. The output of each step within the module is represented in the solid rectangles, in the form of sets of objects, methods, services, directories/files resources, and commands. After that, the automatic PoC generator constructs a ready PoC Android app for each identified risky command. Finally, the automatic verifier verifies whether the PoC app indeed leads to a successful attack, i.e., an unresponsive system.

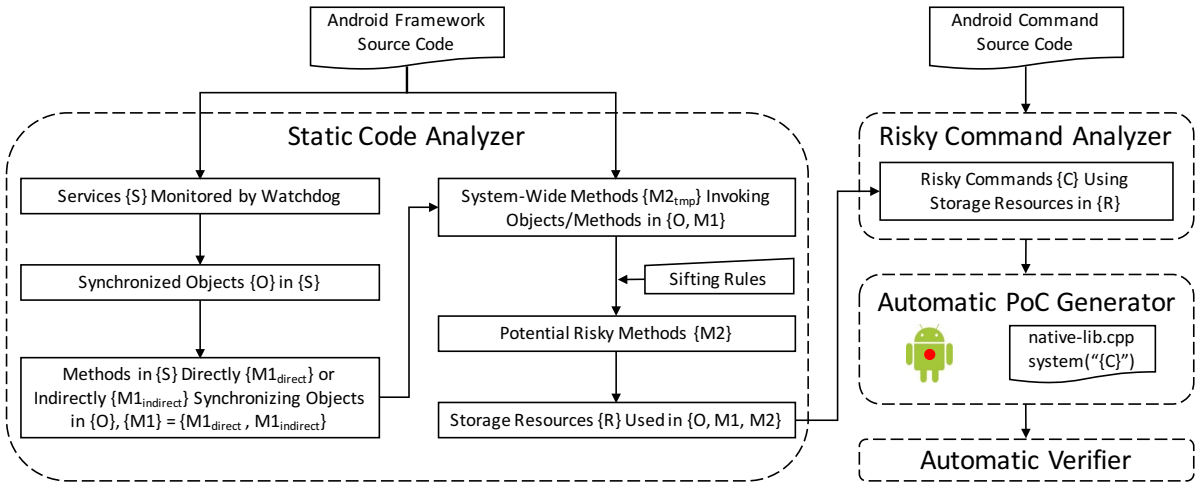


Figure 1: Overview of NativeX.

3.2 Static Code Analyzer

The static code analyzer takes the Android framework source code as input, and outputs a set of system resources, specifically the file system resources, that are used in critical Android system services or other risky framework methods. In order to achieve such target, static code analyzer briefly takes three steps, i.e., identifying the risky objects and methods in the critical system services, sifting-out the risky methods by extending the candidate set to the system-wide, and finally finding out the file system resources used in these risky methods.

First of all, we search for the critical Android system services which are defined as the indispensable services whose failure may cause severe consequences, such as system freeze, crash or reboot. As introduced in Section 2.1, the system services monitored by Android watchdog are so critical that any failure in these services would lead to a watchdog recovery attempt. NativeX utilizes this information to find the critical system services. It finds out the services registered themselves to Android watch dog using `Watchdog.getInstance().addThread()` and `Watchdog.getInstance().addMonitor()`, forming {S} in Fig. 1.

After we have the critical system services, we step further to find the high-risk methods within these critical system services. NativeX treats the methods using a synchronized object as the methods with high risks, because these methods can be starved or delayed while waiting for the synchronized objects. And these starvation and delay could trigger the system reset from the Android watchdog. NativeX firstly identifies such objects set, i.e., $\{O\}$, by recognizing the `synchronized()` blocks in the source code. Following the synchronized objects set $\{O\}$, NativeX further finds out the methods in $\{S\}$ that directly use synchronized objects in $\{O\}$, forming set $\{M1_{direct}\}$. Regarding the indirect methods from $\{S\}$, which are denoted as $\{M1_{indirect}\}$, we traverse the Android source code multiple times to trace back the caller methods. We add every newly-found methods to $\{M1_{indirect}\}$ till there is no more new method. $\{M1_{direct}\}$ and $\{M1_{indirect}\}$ together form $\{M1\}$.

Other than the objects and methods in critical services, invoking points can be from other parts of the Android framework, which highlights the importance of extending the search of risky methods to the system-wide. Different from the candidates in $\{M1\}$, the unresponsive methods that are not monitored by the Android watchdog would not directly cause watchdog triggering the system recovering process. However, if the methods are slowed down significantly while invoking the objects in $\{O\}$ or methods in $\{M1\}$, there is a higher chance that the invoked services trigger the watchdog.

We firstly scan through the whole framework source code to find the methods that are invoking the objects in $\{O\}$ or methods in $\{M1\}$, which is denoted as $\{M2_{tmp}\}$. We then sift out the time-consuming methods in $\{M2_{tmp}\}$ to form a system-wide risky method set $\{M2\}$. NativeX targets the methods that *include loops and have I/O operations in the loop* since loop statements and I/O operations are generally considered to be time-consuming. However, not all the time-consuming methods have loops and I/O operations directly, for example, some methods that gather and update real-time system states are also time-consuming. Despite that our knowledge may not be complete, we would like the result to be as complete as possible. Therefore, to the best our knowledge, we manually add in 10 such time-consuming methods, i.e., `printCurrentState()`, `writeEvent()`, `dumpStackTraces()`, `printCurrentLoad()`, `addErrorToDropBox()`, `updateOomAdjLocked()`, `computeOomAdjLocked()`, `killProcessGroup()`, `isProcessAliveLocked()`, `updateLruProcessLocked()`. Generally, NativeX obtains a set of methods $\{M2\}$ by selecting the methods from $\{M2_{tmp}\}$ using the following two criteria, i) including loops and having I/O operations in the loop; ii) having any of the above 10 methods.

Finally, the static code analyzer outputs the file system resources, i.e., $\{R\}$, used in the above-identified risky candidate sets $\{O, M1, M2\}$.

3.3 Risky Command Analyzer

Risky command analyzer is to locate the Android commands which share the same file resources in {R} with the risky candidates we have identified in the previous step. We believe that due to the weak management of the native process, these native commands are potentially able to be exploited to occupy the file system resources so as to block or slow down the access from the risky objects/methods significantly. This step is also done by analyzing the source code. Differently, it analyzes the Android command source code. Usually, the commands source code is located at `/external/toybox/toys`, `/system/core/toolbox` and `/frameworks/base/cmds`. We filter the Android command source code and obtain the command set {C} including all the commands that have operations in storage resources in {R}.

3.4 Automatic PoC Generator and Verifier

Listing 1: Core code in PoC app template.

```
for(i=0; i<num; i++){
        if(!fork()){
            system("xxx_command");
        }
        if(fork()){
            while(1){
                sleep(1);
            }
        }
    }
```

In the final step, NativeX automatically constructs a PoC application for each command in set {C}. To achieve the automation, we need to manually construct an app template that forks a number of native processes executing a command using a C++ function `system()`. The core code of the template is shown in Listing 1. Then, for each identified command, NativeX generates and builds a new app based on the app template by replacing the parameter of `system()` with the command. The target Android version is set in the `build.gradle` file accordingly so that the app can be compiled to adapt different versions using `gradle` commands, i.e., `gradle assembleDebug` or `gradle assembleRelease`.

Listing 2: Automatic verifier script.

```
adb shell input keyevent 4     // Return
adb shell input keyevent 3     // HOME
adb shell input keyevent 24    // Volume up
adb shell input keyevent 25    // Volume down
adb shell input keyevent 26    // Power
```

The automatic verifier is to verify that the PoC app indeed leads to an unresponsive system. The automatic verifier requires that the testing device is connected via a USB cable to the computer where NativeX is running, and an ADB connection is established between the device and the computer. The verifier installs and launches the PoC app using `adb install` and `adb shell am start`. A list of operations, as shown in Listing 2, are issued to test whether the system can respond properly. The running time for the script under normal condition is around 10 seconds. If the system does not respond timely, e.g., within 60 seconds, or the ADB connection is interrupted, we deem that the system has fallen into an unresponsive state or gone through a reboot.

4 EVALUATION

NativeX is coded using Python and Shell script with 1464 lines of code. It is running on macOS 10.13.6 with 16 GB memory and 2.8 GHz Intel Core i7 processor in this evaluation. We demonstrate how NativeX gradually narrows down the risky candidates and identifies the risky commands. Our evaluation confirms that this vulnerability affects Android versions from 4.2 to 9.0. We further conduct a vulnerability analysis to reveal its root causes. In addition, we present a quantitative evaluation of attack consequences, including device temperature surge, battery degeneration, and computing performance decrease under persistent attack.

4.1 NativeX Evaluation

We run NativeX on the 6 major Android versions ranging from 4.2 to 9.0. We notice that the critical system services monitored by watchdog are changing a lot as shown in Table 1. We demonstrate the NativeX results using the latest Android 9.0 source code. NativeX takes 63.84 seconds to complete the analysis of Android 9.0. As shown in Table 2, in total, we have identified 12 system services that are monitored by the watchdog, where there are 52 synchronized objects and 906 methods directly invoking these objects. In order to find out all the invoking methods, NativeX further traces back along the call graph and obtains 410 methods indirectly invoking these objects, which takes five layers tracing backward. Following the critical objects and methods (i.e.,{O} and {$M1$}), the NativeX static code analyzer extends to the system-wide and collects 77 methods (i.e.,{$M2$}) that are not monitored by watchdog but using objects and methods in {O} and {$M1$}. Finally, nine root storage resources (i.e., {R}) are identified, i.e., `/data`, `/dev`, `/mnt`, `/proc`, `/product`, `/storage`, `/sys`, `/system`, `/vendor`.

Atop these system storage resources, risky command analyzer filters the Android command source code and obtains the risky commands as shown in Table 3. Due to the page limit, Table 3 only lists the numbers for each version[1]. A small portion of the total commands are recognized as risky, for example, out of the 269 commands found in the Android 9.0 command source code, there are 50 risky commands that have operations in the 9 critical storage resources. Readers may have noticed that not all of the risky commands identified from the Android command source code are available in the Android system. More about this interesting observation is explained in Section 4.2. Finally, the automatic PoC generator constructs a PoC app for each identified command. The number of native processes that the PoC app is to spawn is set to a large enough number 10,000. In fact, the experiments later will show that a successful attack only takes 80 to 2,800 processes depending on the device specifications.

The PoC apps are verified on 19 smartphones and 2 smart TV boxes. We manage to cover all the 6 major versions we analyzed using Google Nexus and Pixel series. Besides Google phones, we also test other models running vendor-customized Android systems, such as Samsung, Huawei, etc. In addition to smartphones, there are

[1]Please refer to the following link for detailed risky commands from the 6 major Android versions, https://goo.gl/Qjtmbp

Table 1: The critical system services monitored by watchdog in different Androids versions.

Android OS Version	Monitor Services	Number
Android 4.2	ActivityManagerService, WindowManagerService, PowerManagerService, InputManagerService, NetworkManagementService, MountService	6
Android 5.1	ActivityManagerService, WindowManagerService, PowerManagerService, InputManagerService, NetworkManagementService, MountService, PackageManagerService, MediaRouterService, MediaSessionService, MediaProjectionManagerService	10
Android 6.0.1	ActivityManagerService, WindowManagerService, PowerManagerService, InputManagerService, NetworkManagementService, MountService, PackageManagerService, MediaRouterService, MediaSessionService, MediaProjectionManagerService	10
Android 7.1.1	ActivityManagerService, WindowManagerService, PowerManagerService, InputManagerService, NetworkManagementService, MountService, PackageManagerService, MediaRouterService, MediaSessionService, MediaProjectionManagerService, TvRemoteService	11
Android 8.0	ActivityManagerService, WindowManagerService, PowerManagerService, InputManagerService, NetworkManagementService, StorageManagerService, PackageManagerService, MediaSessionService, MediaRouterService, MediaProjectionManagerService, TvRemoteService	11
Android 9.0	ActivityManagerService, WindowManagerService, PowerManagerService, InputManagerService, NetworkManagementService, StorageManagerService, PackageManagerService, MediaSessionService, MediaRouterService, MediaProjectionManagerService, TvRemoteService, PermissionManagerService	12

Table 2: Synchronized objects and their invoking methods in the critical services monitored by the watchdog.

$\{S\}$	$\{O\}$	$\{M1_{direct}\}$	$\{M1_{indirect}\}$
ActivityManagerService	16	324	137
InputManagerService	6	27	8
MediaProjectionManagerService	1	10	8
MediaRouterService	2	16	12
MediaSessionService	1	22	8
NetworkManagementService	6	29	12
PackageManagerService	6	217	142
PermissionManagerService	2	18	8
PowerManagerService	3	55	11
StorageManagerService	4	36	19
TvRemoteService	1	2	1
WindowManagerService	4	150	44
Total	**52**	**906**	**410**

two smart TV boxes running Android 4.4 and 6.0. Table 4 shows our testing results. All test devices with corresponding clean systems are successfully attacked by all of the attack PoC applications of that version. Generally, the attack PoC apps take effect within 5 seconds despite small differences among different testing devices. It means that the vulnerability is, in fact, affecting a very large range of Android devices from 4.2 to the latest 9.0 which contribute to 99.7% of the whole Android devices [9]. Section 5 provides more details about PoC apps and the attack demos which are publicly available on Google Play and Youtube. We also test the minimum number of native processes that a successful attack requires on specific devices. We can see from the column "Min # Persistent" and "Min # Unsupported" in Table 4 that the minimum numbers vary a lot from 80 to 2,830 on different devices. These numbers provide us a practical threshold when we discuss the defense approaches later in Section 6. We can see that the "Min # Persistent" values are much less than "Min # Unsupported" values, which are explained in detail in Section 4.2.

4.2 Vulnerability Observation and Analysis

During the evaluation, there are two interesting observations.

Observation #1. Readers may have noticed that in Table 3, not all of the risky commands identified from the Android command source code are available in the Android system. According to our experiments, even though the command source code indeed exists, some commands cause the command-not-found error. However, we found that all of the identified commands can achieve a similar attack consequence.

In order to figure out the underlying reason, we manually analyze the attack trace and reveal that the underlying causes are different despite the undifferentiated attack consequences. The analysis is based on the Android watchdog log information. The Android watchdog dumps the traces which help us to identify the causes of the system reset. The error messages vary in different situations. The error message can be "blocked in handler" or "blocked in monitor". The first means there is an overtime situation occurs, while the latter indicates a deadlock. The causes are explained according to different command categories.

Persistent commands. Among the commands NativeX has identified, there are some persistent commands that reside in memory after started, for example, `top`. Under the attack using persistent commands, the error messages are non-deterministic. There are both "block in monitor" and "block in handler" in the log, which means there are sometimes deadlock and sometimes overtime situations. When the PoC application invokes such persistent commands, the persistent commands directly occupy the shared system storage resources that only support the mutually exclusive operation, i.e., some system directories, so that the critical system

Table 3: Risky command statistics in Android versions from 4.2 to 9.0.

Android OS Version	Storage resources	# Risky commands	# Available in Android	# Total commands
Android 4.2	/data, /proc, /system, /sys, /d	15	14	96
Android 5.1	/data, /proc, /system, /sys, /d, /vendor	15	14	91
Android 6.0.1	/data, /proc, /system, /sys, /vendor, /storage, /dev, /mnt	52	30	253
Android 7.1.1	/data, /proc, /system, /sys, /vendor, /storage, /dev, /mnt	50	28	257
Android 8.0	/data, /proc, /system, /sys, /vendor, /storage, /dev, /mnt	55	31	262
Android 9.0	/data, /proc, /system, /sys, /vendor, /storage, /dev, /mnt, /product	50	29	269

Table 4: Vulnerability verification results. The testing devices include 19 smartphones and 2 smart TV boxes. "Min # Persistent" and "Min # Unsupported" indicate the required number of processes to succeed an attack using a persistent command and an unsupported command, respectively.

Device Model	System Version	Affected	Min # Persistent	Min # Unsupported	x times more
Pixel 2	Android 9.0 and 8.1	✓	1510	5530	3.66
Pixel	Android 8.1	✓	1280	5090	3.98
Pixel XL	Android 8.1 and 7.1	✓	1280	5090	3.98
Nexus 6P	Android 7.1	✓	840	3600	4.29
Nexus 5X	Android 6.0.1	✓	560	3490	6.23
Nexus 5	Android 5.1 and 4.2	✓	450	2210	4.91
Xiaomi 6	MIUI 10.0 (Android 8.0)	✓	1640	6270	3.82
Xiaomi 4	MIUI 8 (Android 6.0.1)	✓	480	2370	4.94
Xiaomi Note	MIUI 8 (Android 6.0.1)	✓	750	3250	4.33
Huawei nova 2	EMUI 8.0 (Android 8.0)	✓	2830	7600	2.69
Huawei P8	EMUI 3.1 (Android 5.0)	✓	800	3520	4.40
Smartisan Pro2	Smartisan OS 6.1.1 (Android 7.1.1)	✓	1170	3960	3.38
Smartisan Pro	Smartisan OS 3.7.3 (Android 7.1.1)	✓	960	3040	3.17
Samsung Galaxy Note4	Android 6.0.1	✓	720	2490	3.46
Meizu Meilan Note3	Flyme 6.1 (Android 5.1)	✓	600	2020	3.37
Meizu Meilan2	Flyme 4.5 (Android 5.1)	✓	450	1770	3.93
Coolpad 7270	Android 4.2.2	✓	80	500	6.25
Tencent 1v	Android 6.0	✓	350	2100	6.00
Skyworth T2	Android 4.4	✓	190	1680	8.84

services are starved and blocked. Also, we suspect that sometimes the attack leads to the computing resource exhaustion due to the large number of running processes. The log message from the Android watchdog depends on which situation triggers the watchdog first.

Non-persistent commands and unsupported commands. Apart from the persistent commands, the rest commands are non-persistent commands and the unsupported commands. We classify them as one category because they share the same underlying reason in causing a system failure. Both of them cannot occupy system or device resources for a long time. The non-persistent commands exit once the task is completed, and the unsupported commands exit once the command is alerted unsupported. Taking advantage of the weak management of the native process, the PoC app can fork a large number of the processes running such commands. Under the attack using this category of commands, the error messages are generally "block in handler", which means the Android watchdog monitors the overtime situation first. We conjecture that a large number of processes lead to the surge of the number of folders and files in the storage resources, e.g., `/proc`, which slows down the access from critical system services, e.g., traversing the directories to gather process information. And most importantly, these processes exhaust the device computing resources, which leaves limited computing resources to critical system services. Both situations make the operation in critical services exceed the watchdog preset time, and finally trigger the recovering process.

To better understand the different effects of the persistent commands and non-persistent/unsupported commands, we compare the required number of processes to launch a successful attack using a risky persistent command and an unsupported command, respectively. The results can be found in the last three columns in Table 4. We can see that it requires much fewer processes to launch an attack using persistent comparing to using unsupported command. The last column shows how many times more processes it requires using unsupported commands than using persistent commands. This experiment helps to understand the different contributions of the two different types of commands. It also indicates that occupying the system resources is more effective than exhausting the device computing resources when launching such an attack.

Observation #2. The second interesting observation is watchdog's failure in recovering the system from the attack in Android 8.0 and 9.0. Android is able to reboot itself from the freeze in versions lower than 8.0, however, Android systems with version 8.0 and 9.0 are prone to be frozen forever until a manual reboot by long pressing the power button. Watchdog is the last defense to ensure the running of the Android system. In case any failure that delays the operation in the critical service to make it exceed the preset timer, watchdog kills the whole system server to go through a soft-reboot. One would wonder why it kills the whole system server instead of the relevant service process. In fact, it has been pointed out that such design can be utilized to maliciously shut down the system server by tricking the watchdog to reboot the Android (version 4.0 to 5.0.2) userspace including the system server, the Zygote and other processes [22]. We speculate that from Android 8.0, there is an improvement in such design that the watchdog thread no longer blindly kills the whole system server. This speculation is based on the system log information which shows tremendous effort in restarting the dead critical services but all are unfortunately end up with "start timeout". This effort is not sufficient to recover the system from our attack, which means that the frozen system is out of service until the battery is empty unless there is human interruption forcing a reboot. If the device does not have a battery, such as the TV box, it requires to unplug the device to shut down the system forcibly. This observation motivates us to explore more about attack consequences later in Section 4.3, which reveal fruitful findings.

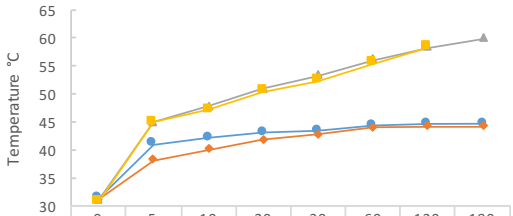
4.3 Quantitative Attack Consequences

This section demonstrates the quantitative measurement of the attack consequences exploiting the identified vulnerability. We select Google phones Pixel and Pixel 2 as the testing devices according to the Google vulnerability reward program.

4.3.1 Temperature Surge. Different from the existing DoS attacks on Android systems that would lead to an automatic system recover [22], we notice that the late Android versions 8.0 and 9.0 fail in automatically recovering from our attack. This observation motivates us to explore the attack consequence due to the long-time system freeze. One of the immediate effects is that the device temperature significantly rises compared to that under the normal condition. In this subsection, we conduct experiments to measure the surface temperature of the Google Pixel under both normal conditions and attack conditions.

Under the attack condition, the testing device runs one of the PoC applications that would lead to a system freeze, whereas under the normal condition, it runs an endless loop of AES encryption and decryption which simulates the processing load of normal user operations. For each condition, we measure both temperatures when the device is connected to a charger and when it is not. All the other running settings are identical for both normal and attack conditions, for example, the screens are both lighted and the air temperatures while testing are both 28 °C. The temperature of the device surface is measured using a contactless industrial infrared thermometer (Hong Kong Xima AS380). We record the device temperature at its front center or back center whichever is higher.

We run the test for 3 hours and record the temperatures during the whole test. Each test condition is repeated 10 times for us to obtain an average temperature value. The results are shown in Figure 2. We can see that under the normal condition, the device temperature maintains a stable level during the whole test and reaches no more than 45 °C even if the device is constantly running heavy computing tasks, i.e., AES encryption and decryption. However, under the attack condition, the device temperature continuously rises as time goes on and reaches as high as 60 °C after 3 hours of testing. The measurement at 180 minutes is not available for "attack not charging" condition, because the attack has drained the battery before reaching 180 minutes. Regarding the influence of the charging, given the same normal/attack condition, the temperatures for charging and not charging are generally similar, which implies the influence to device temperature from charging is negligible.



	0	5	10	20	30	60	120	180
Normal Charging	31.26	41	42.02	43.08	43.38	44.3	44.66	44.64
Normal Not Charging	31.14	38.1	40.1	41.84	42.72	44	44.14	44.14
Attack Charging	31	44.92	47.82	50.9	53.28	56.18	58.34	59.84
Attack Not Charging	30.7	45	47.2	50.46	52.4	55.5	58.3	

Figure 2: Device temperature measurements.

4.3.2 Persistent Battery Degeneration. Since Android cannot automatically recover from the attack freeze, it is feasible to observe the gradual impact on the device from the long-term attack. In this experiment, we attack the charging devices for 7 hours per day (simulating the scenario that the attack happens when the device is charging overnight) and 20 days in total. After every 7-hour attack, we reboot the system to stop the attack application and start our battery life measurement experiments. For the rest of the day, we leave the device standby in normal condition.

The battery life is measured as the time it takes from the full battery charge to auto power off. We keep the screen on and run the same application, which has an endless loop of AES encryption and decryption, on all the testing devices to simulate the user usage. We have two sets of devices under test, one set is under attack and the other is under normal use. Each set has two models, i.e., Google Pixel and Google Pixel 2. In order to provide a clear view of the comparison, we normalize the battery life to [0,1] for each set, i.e., Pixel and Pixel 2, respectively. The results are demonstrated in Figure 3. After 20 days of normal use, the battery lives of the two device models do not change much. In comparison, it can be seen that battery life is reducing gradually since the attack is launched. After 20 days of the attack, the battery life of Pixel reduces 13% (29 minutes) and Pixel 2 reduces 22% (54 minutes), which indicates that the battery life is reduced significantly under continuous attacks.

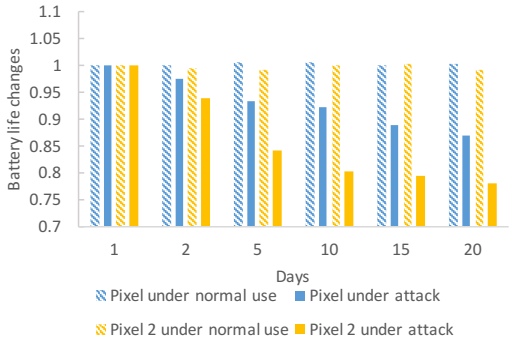
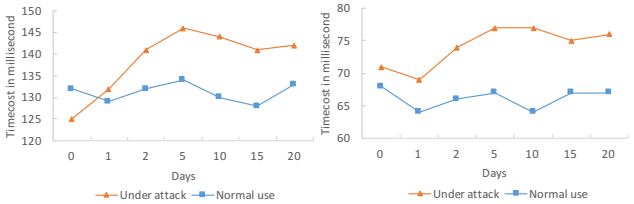


Figure 3: Battery life changes under attack or normal use for 20 days. The vertical axis is normalize to [0,1] for two devices, respectively.

4.3.3 Persistent Computing Performance Decrease. This experiment is to measure the potential impact on the computing performance when the device is under attack for a period of time, i.e., 20 days in our setting. The setting of this experiment is the same as the previous battery degeneration experiment in Section 4.3.2. The test devices are rebooted before every measurement. The computing performance is measured by the processing time of running a program piece. The shorter the time it takes, the better the performance is. The device is charging during the performance measurement to avoid the influence of the battery level. In our experiment, we run AES encryption/decryption 1000 times and measure its time cost. Figure 4 illustrates the results of the two models, i.e., the Pixel and Pixel 2, after days of attack and normal use. We can see from Figure 4 the fluctuation in the processing time for both models under normal use is tiny. In contrast, when the device is under attack, the processing time increases generally as the attack carries on. The decrease in milliseconds of running the test program piece may be negligible to humans, however, it is worth noting that the decrease is about 10% in 20 days. Though we have observed the decrease in computing performance, it is difficult to pinpoint the root of the decrease in computing performance because the computing performance is determined by a number of factors such as CPU and RAM.



(a) AES 1000 times on Pixel. **(b) AES 1000 times on Pixel 2.**

Figure 4: The device computing performance under attack and normal use measured by the processing time of AES encryption/decryption (1000 times) on Pixel and Pixel 2.

5 POC ATTACKS AND HAZARDS

In this section, we analyze the potential hazard of the identified vulnerability by engineering several representative PoC attacks, including the attack scenario, the design of the attack, as well as the consequences, which demonstrate the easily-exploitable nature of the identified vulnerability. The PoC attack apps have successfully passed the security screening mechanism in the Google Play [2].

5.1 DoS Attacks against Android System

Besides dominating 85% of smartphone market share [12], Android systems are also widely used in IoT devices and critical tasks, such as medical devices and on-vehicle/aircraft devices. Any failure, no matter temporary or long-term, in the critical tasks could lead to severe consequences which could be a matter of life and death in some of the applications. Unfortunately, one straightforward way to exploit the vulnerability identified by NativeX is to perform DoS attacks. The critical tasks performed in the system could be suspended by a system freeze or a system soft-reboot, which makes the identified vulnerability an ideal tool for sabotaging attackers especially the ransomware attackers. The severe consequences of DoS attacks may seem alike, the DoS attacks leveraging on our newly-discovered vulnerability is stealthier since it does not require any permission above the "normal" level.

Attack PoC design and attack consequences. Android broadcasts an Intent message including a system-defined action string `android.intent.action.BOOT_COMPLETED` when the system has finished booting either from a soft-reboot or a complete restart. The PoC is designed to listen to the booting broadcast by registering a broadcast receiver. A sabotaging attacker can start the attack once the PoC app gets notified that the system has finished booting. For a ransomware attacker, he/she can instead pop up an activity with his/her ransom payment information before the launch of the actual attack. The activity can also be designed in a manner that the PoC app would exit if the device owner has paid the ransom, otherwise launch the attack. Both sabotaging attack and ransomware attack will not allow the user to interact with the system[3].

5.2 DoS Attacks against Android App

By exploiting the identified vulnerability, an attacker is able to disturb the functioning of a victim app that has already been installed on the device. This is a type of highly targeted attack. The DoS attack is triggered only when the user is performing a certain operation or interacting with a certain app. It is difficult to detect the source of such an attack since it is triggered by the legitimate behaviors of the victim app and it requires no specific permission for the attack app to receive the trigger timing. From the user's perspective, it would appear as the system freezes or reboots once the user intends to interact with the victim app. In this case, an attacker, e.g., a vicious app competitor, may purposely launch such an attack against its rival to frame the rival app and further mislead the user to uninstall the rival app.

[2]The PoC apps are available in the following link: https://play.google.com/store/apps/details?id=com.hou.ndkdemo. We clearly mark this app as an academic PoC for a research paper to avoid misleading normal users.
[3]The DoS attack against the Android system demo is available at https://youtu.be/VmvJW-6rGmQ

Attack PoC design and attack consequences. Instead of listening to the system booting broadcast, this PoC listens to the broadcast from the victim app. It requires that the target app sends any broadcast that can be received by other third-party apps, which is in fact very common in Android apps. Apps usually send out implicit broadcasts to notify other apps on the same device of its operation or status, such as the app launch and the user login. The PoC app thus can register a receiver to receive the broadcast from the target app. If the broadcast is protected by permission (not at the signature level), the receiver app needs to apply for the permission and easily tricks the user to approve the permission application [34]. Otherwise, no permission is required. The PoC app is designed in the way that it launches the attack once its receiver receives the broadcast from the victim app.

We have conducted a preliminary experiment to verify the above attack design using popular apps with billions of downloads. The apps and their broadcasts that can be used by the attack PoC are listed in Table 5. The metadata of the app is from the Tencent App Store which is one of the largest app stores in China. We only list the broadcasts that are triggered by user operations in Table 5. By listening to different broadcast actions, the attack can be launched at different points, for example, when the user is in the login process. When the attack PoC launches DoS attack upon receiving such broadcasts, it gives users the impression that it is some function in the victim app that causes the system to freeze and reboot, such as logging in the app. This attack disturbs the users when they are interacting with the victim app, and further mislead the users to uninstall the victim app[4].

5.3 Physical Harm to Users

The experiments in Section 4.3.1 reveal that the temperature of the device under attack reaches as high as 60 °C when the environment temperature is 28 °C. However, the environment temperature could be much higher than our controlled environment temperature 28 °C. For example, one case could be that an Android smartphone is in an enclosed space such as a car parking in the summer sun (the in-car temperature can reach up to 70 °C, after parking in the sun on a 38 °C day [4]), or surrounded by daily thermal insulation materials such as under a pillow in the bed. In such cases, the device temperature under attack might be even higher than 60 °C which may not be immediately but potentially cause physical harm to users.

Attack PoC design and attack consequences. This PoC app is designed to overheat the victim Android device to cause any harm to the device or users. Android provides temperature sensor APIs for apps to acquire temperature. The PoC app first creates an instance of the `SensorManager` and use the sensor manager instance to further get an instance of a temperature sensor. After registering a sensor listener, the PoC app can start handling the sensor data in the `onSensorChanged()` callback. The PoC app has a preset temperature threshold, reaching which will trigger the attack. We have not conducted repeat attack experiments to get an average device temperature as we have done in Section 4.3.1 to avoid any harm from the overheat. We do have a test when the environment temperature is 42.9 °C which is the temperature on a hot summer day. After 200 minutes attack, despite the CPU throttling technique in modern CPUs, the device temperature still reaches 70.4 °C [5]. The temperature is far beyond the suggested charging (0 °C to 45 °C) and discharging temperature (-20 °C to 60 °C) of the lithium-ion batteries [2]. Not to mention the degeneration to the battery [33], the overheat is also able to suffer users from burns [8].

6 DEFENSE APPROACHES

One way to stop this type of attack is to remove the attack app from the system. Android safe mode is such a special runtime environment where no third-party app is allowed to run so that users can uninstall any identified attack app without any attack interference. Unfortunately, it is usually a challenging task for ordinary users to recognize the attack app. The attack app can disguise itself as a functional app. It may launch attacks intentionally on various occasions. Other than the occasions we explained in Section 5.1 and Section 5.2, the attack apps can be triggered by other permission-free indicators, such as a preset time, battery level, or system running time, which makes it difficult for users to identify the sources of such attacks. Alternatively, a factory reset, which may erase all the useful applications and data, can stop the attack, however, at the cost of unnecessary labor work to restore the previous system settings and user data. Therefore, it is preferred to solve the problem by amending the management mechanism for native processes instead of burdening users with the professional security challenge.

Due to the weak management of the native processes, any app can starve or slow down the critical system services by invoking the Android commands that share the system resources with these services in its native processes. Native programming is an effective method to achieve extra performance as well as to defend the core algorithm against reverse engineering. The native API used in the attack PoC app, i.e., `system()`, is widely used in command-line shell apps. It is recommended to regulate the usage of such native APIs instead of crudely forbidding them. In this section, we briefly discuss the possible defense approaches from the Android system angle.

Introduce Permission Control. Unlike the established permission control in the framework programming, there is no such control on the NDK programming. As a consequence, any application can invoke any native API without any restriction. We suggest introducing permission control to the native APIs, which follows the uniform security mechanism in the framework layer. Similarly, the app that intends to issue Android commands via framework APIs, e.g., `Runtime.getRuntime().exec()`, or native APIs, e.g., `system()`, shall request for the corresponding permission before such operation. On one hand, permission control is able to restrict the usage of the target APIs/resources by informing the system as well as the users of such access. On the other hand, it provides extra permission information to conduct malware detection using the traditional permission-based approach.

On the downside, users may not understand the concept of permission controlling the execution of commands, which may require the specification of such permission to be intuitive. Moreover, introducing extra permission control to the native layer,

[4]The DoS attack triggered by Taobao login broadcast is demoed in the link: https://youtu.be/HJR_r4DH-DE

[5]The overheat attack demo video is available at: https://youtu.be/kZbCWNlKARA

Table 5: DoS attack against popular apps. The listed broadcast actions can be used to trigger the DoS attacks.

App	Version	Downloads	Last updated	Broadcast actions that are triggered by user interactions
Taobao	V8.10.0	2.8 billion	30/7/19	TAOBAO_DELAY_START_LOGIN
Wechat	V7.0.6	7.9 billion	26/7/19	com.tencent.mm.plugin.openapi.Intent.ACTION_REFRESH_WXAPP
QQ	V8.1.0	9 billion	30/7/19	com.tencent.mobileqq.msf.startmsf
Sina Weibo	V9.8.0	2.3 billion	31/7/19	sina.weibo.action.UPDATED com.sina.weibo.guardunion.NEW_DATA com.sina.weibo.photo.action.UPDATE_VIDEO_CONFIG
iQiyi	V10.7.0	2.5 billion	23/7/19	com.iqiyi.hotchat.user.login org.qiyi.video.module.action.REGISTRY com.aiqiyi.shortvideo.database
Youku	V8.0.5	1.9 billion	23/7/19	com.youku.service.download.ACTION_DOWNLOAD_SERVICE_CONNECTED com.youku.action.GET_INIT_DATA_SUCCESS com.youku.phone.home.loadfinish
NetEase Music	V6.3.0	200 million	29/7/19	com.netease.cloudmusic.action.UPDATE_CRASH_HANDLER_USERID com.netease.music.action.STAR_MUSIC
Ctrip	V8.8.0	330 million	30/7/19	ctrip.location.coordinate.success

which is brand new to the Android system, may bring extra performance overhead to the system. And similar to every previous permission update, it would require the app developers adapting their apps to the new permission update.

Restrict the Number of Native Processes. According to our vulnerability analysis, an app is able to spawn an unlimited number of native processes so that it can starve the system service or slow down the entire system to freeze the system or force a system reboot. A straightforward but effective approach is to control the maximum number of native processes that an app can spawn so that the attack app cannot have enough number of running native commands to continuously occupy the shared system resources or exhaust the device computing resources. It is a usual way adopted by Android, for example, the maximum number of `Toasts` that an app can pop is 50 [34], and the maximum number of active locks an app can create is also 50 [21].

We conducted a brief study by randomly selecting 50 apps from the top 500 apps in Google Play and counting up the native processes in these apps. According to the statistics, the number of native processes that an app has is normally less than 10. Therefore, it is relatively easy to set a reasonable threshold of the maximum number of native processes that a third-party app can spawn. According to Table 4, the minimum number of native processes required by a successful attack is 80. It is on Coolpad 7270 which is an old model released in 2012. Newer models have much bigger figures of the minimum native process numbers. Therefore, fifty would also be a conservative limit in this case based on our extra defense experiments which demonstrate the PoC app is far from success on any of our testing device if its number of native processes is set to 50. For trusted system apps, the threshold can be loosened if necessary.

Regarding the implementation, Android is ready to impose such a limit on the number of processes an app can have using `SetRLimits()` which is based on the Linux `setrlimit()`. Android applies SetRLimits() to all the child processes forked from the Zygote, including all the app processes. We investigate the maximum number of processes that a third-party app can have. The limit varies in different versions and models, for example, on our experiment devices, the limits are 14,096 in Android 7.1, 22,097 in Android 8.1, and 21552 in Android 9.0. The limit numbers are all much larger than the number of processes required by an attack. Though the existing limit cannot effectively prevent the attack in our paper, we could readily configure the parameter of this function to provide a tighter and reasonable limit.

Compared to the permission-based solution, restricting the number barely brings any burden to the Android system. It requires no specific adaption from either app developers or app users. This may explain why it is preferred by system patches and previous research works [34][21]. A potential problem may come from the exact limit number. Even though the statistic analysis on the top-ranked apps shows that 50 would be a suitable number that would not affect these apps, it may require cautious confirmation of such statistical observation.

7 RELATED WORK

`Android DoS Attacks.` As a prevailing mobile system, Android has been widely explored by researchers to exploit any vulnerability in the system to launch DoS attacks. The attack points can be generally categorized into two layers according to the Android system architecture. One category of attack points are from the Android framework layer, which is based on the exploits of system services or system components. For example, there are DoS attacks exploiting the lack of input validation [16], the inconsistent security enforcement within the Android framework [29], the design trait in the concurrency control mechanism of the system server [22], the vulnerability in the call back mechanism in system services [31], as well as the lack of access control and memory usage limit in various ION heaps [35]. Also, there are attacks that can force the process

abort and trigger the system reboot via IPC flooding [20], JGR (JNI Global Reference) exhaustion [21], and Toasts flooding [24]. The other attack point category is to launch DoS attacks from the native layer. Armando et al. take advantage of a security breach in permission management of the Zygote socket so that any process can request the Zygote process to spawn unlimited new processes and finally exhaust the system resources [13].

The consequences of a DoS attack may seem alike, the root causes of the vulnerability are totally different. The vulnerability identified in our work distinguishes itself in a way that exploits the weak management of the Android native process. And the showcased attacks represent the first exploration taking advantage of the legitimate Android commands as attack vectors. Besides DoS against the system, we also design the attack against target apps, which has not been explored before to the best of our knowledge. Moreover, we quantitatively evaluate the attack consequences caused by this vulnerability, i.e., temperature surge, battery degeneration, and computing performance decrease.

`Android Static Analysis.` The open-source nature of Android enables the static source code analysis to be one of the most prominent analysis methods in analyzing the system itself and its apps. In the Android application analysis area, the previous work uses static analysis to detect the component hijacking vulnerabilities in Android apps [26], identify the security and privacy issues in Android apps [25] and in advertisement libraries used in Android apps [18], and reveal the capability leaks in stock smartphones [19]. There are also general application analysis frameworks, such as FlowDroid [14], Amandroid [32] and EPICC [27]. FlowDroid is an effective static taint analysis framework for Android applications, which can achieve precise context, flow, field, and object-sensitive taint analysis [14]. Amandroid is another static application analysis tool for determining points-to information for all objects in Android applications and context-sensitive functions across Android applications components [32]. Besides analyzing apps, there are also static analysis tools for system vulnerability analysis based on the Android source code [20][21][22][29][35]. SUSI is proposed to leverage a machine-learning guided approach to classify and categorize sources and sinks in the framework layer and pre-installed apps [28]. Backes et al. build a static runtime model of the Android framework to analyze its internals and find the high-level protected resources so as to reveal the influence on the platform security and user privacy [15]. We can see that the static analysis achieves a large range of purposes. Thanks to statistic analysis, NativeX can highly target the risky points in the Android system, saving a lot of labor inspection work.

8 CONCLUSION

This work reveals a new vulnerability existing in a wide range of Android versions from 4.2 to 9.0 due to the weak management of native processes. Exploiting this vulnerability, we take the first exploration using legitimate Android commands in native processes as attack vectors, such that any third-party app without any permission can freeze the system or force the system to reboot. We design an integrated analyzing and testing tool named NativeX, to identify the risky Android commands, construct PoC apps, and verify the effectiveness of the PoC apps. We quantitatively measure the attack consequences of attacks exploiting this vulnerability, including device temperature surge, battery degeneration, and computing performance decrease. We further present three representative attacks to explain the hazards to the Android system, Android on-device apps, and device users. Finally, we conduct vulnerability analysis and reveal the root causes of the vulnerability, based on which, we discuss the defense approaches against such vulnerability.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (NSFC project U1836113 and U1836117).

REFERENCES

[1] 2015. 5 airplane entertainment systems that will make you forget youâĂŹre flying coach. https://www.digitaltrends.com/cool-tech/5-best-airlines-for-in-flight-entertainment/.
[2] 2017. BU-410: Charging at High and Low Temperatures. https://batteryuniversity.com/learn/article/charging_at_high_and_low_temperatures.
[3] 2017. Google's Android now powers more than 2 billion devices. https://www.cnet.com/news/google-boasts-2-billion-active-android-devices/.
[4] 2018. How Long Does It Take a Parked Car to Reach Deadly Hot Temperatures? https://www.livescience.com/62651-how-hot-cars-get.html.
[5] 2018. Volvo's native Android Auto system. https://www.youtube.com/watch?v=SoGrE6t4ejQ.
[6] 2019. Android Automotive. https://source.android.com/devices/automotive.
[7] 2019. Lock. https://developer.android.com/reference/java/util/concurrent/locks/Lock.
[8] 2019. Thermal burn. https://en.wikipedia.org/wiki/Thermal_burn.
[9] 2020. Distribution Dashboard. https://developer.android.com/about/dashboards/.
[10] 2020. DJI DEVELOPER TECHNOLOGIES. https://developer.dji.com/.
[11] 2020. ingenico GROUP. https://ingenico.us/mobile-solutions#tablet-pos.
[12] 2020. Smartphone OS Market Share. https://www.idc.com/promo/smartphone-market-share/os.
[13] Alessandro Armando, Alessio Merlo, Mauro Migliardi, and Luca Verderame. 2012. Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In *IFIP International Information Security Conference*. Springer, 13–24.
[14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
[15] Michael Backes, Sven Bugiel, Erik Derr, Patrick D McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis.. In *USENIX Security Symposium*. 1101–1118.
[16] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. 2015. Towards analyzing the input validation vulnerabilities associated with android system services. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 361–370.
[17] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *23rd USENIX Security Symposium (USENIX Security 2014)*. 1037–1052.
[18] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 101–112.
[19] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic detection of capability leaks in stock android smartphones.. In *NDSS*, Vol. 14. 19.
[20] Yacong Gu, Yao Cheng, Lingyun Ying, Yemian Lu, Qi Li, and Purui Su. 2016. Exploiting android system services through bypassing service helpers. In *International Conference on Security and Privacy in Communication Systems*. Springer, 44–62.
[21] Yacong Gu, Kun Sun, Purui Su, Qi Li, Yemian Lu, Lingyun Ying, and Dengguo Feng. 2017. JGRE: An Analysis of JNI Global Reference Exhaustion Vulnerabilities in Android. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 427–438.
[22] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. 2015. 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*. ACM, 1236–1247.
[23] Ryan Johnson, Mohamed Elsabagh, Angelos Stavrou, and Vincent Sritapan. 2015. Targeted DoS on android: how to disable android in 10 seconds or less. In *2015*

10th International Conference on Malicious and Unwanted Software (MALWARE). IEEE, 136–143.

[24] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. 2010. These aren't the permissions you're looking for. *DefCon* 18 (2010).

[25] Kangjie Lu, Zhichun Li, Vasileios P Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. 2015. Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting.. In *NDSS*.

[26] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 229–240.

[27] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. *Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis* (2013).

[28] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks.. In *NDSS*.

[29] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. 2016. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework.. In *NDSS*.

[30] Stephan Spat, Kevin Theuermann, Bernhard Höll, Peter Beck, and Thomas R Pieber. 2014. Development, integration and operation of mobile, Android-based medical devices in hospitals: Experiences from the GlucoTab® system. In *Wireless Mobile Communication and Healthcare (Mobihealth), 2014 EAI 4th International Conference on*. IEEE, 128–131.

[31] Kai Wang, Yuqing Zhang, and Peng Liu. 2016. Call me back!: Attacks on system server and system apps in android through synchronous callback. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 92–103.

[32] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1329–1341.

[33] Xiao-Guang Yang and Chao-Yang Wang. 2018. Understanding the trilemma of fast charging, energy density and cycle life of lithium-ion batteries. *Journal of Power Sources* 402 (2018), 489–498.

[34] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. 2016. Attacks and defence on android free floating windows. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 759–770.

[35] Hang Zhang, Dongdong She, and Zhiyun Qian. 2016. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1663–1674.