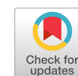
Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A survey of Android exploits in the wild



Huasong Meng *, *Vrizlynn L.L. Thing*, *Yao Cheng*, *Zhongmin Dai*, *Li Zhang*

Institute for Infocomm Research, Agency for Science, Technology and Research, Singapore, 138632, Singapore

ARTICLE INFO

Article history:
Received 31 January 2018
Accepted 26 February 2018
Available online 8 March 2018

Keywords:
Android
Mobile security
Privilege escalation
Exploit
Survey

ABSTRACT

The Android operating system has been dominating the mobile device market in recent years. Although Android has actively strengthened its security mechanisms and fixed a great number of vulnerabilities as its version evolves, new vulnerabilities still keep emerging. Vulnerability exploitation is a common way to achieve privilege escalation on Android systems. In order to provide a holistic and comprehensive understanding of the exploits, we conduct a survey of publicly available 63 exploits for Android devices in this paper. Based on the analysis of the collected real-world exploits, we construct a taxonomy on Android exploitation and present the similarities/differences and strength/weakness of different types of exploits. On the other hand, we conduct an evaluation on a group of selected exploits on our test devices. Based on both the theoretical analysis and the experimental results of the evaluation, we present our insight into the Android exploitation. The growth of exploit categories along the timeline reflects three trends: (1) the individual exploits are more device specific and operating system version specific; (2) exploits targeting vendors' customization grow steadily where the increase of other types of exploits slows down; and (3) memory corruption gradually becomes the primary approach to initiate exploitation.

© 2018 Elsevier Ltd. All rights reserved.

1. Introduction

Smart mobile devices are indispensable in people's lives nowadays. Along with the development of mobile technology and the prevalence of Internet services, smart mobile devices become the principal digital assistant that people use for information acquiring, instant messaging, online socialization, Internet financing and other Internet services. The market share of devices with Android operating system keeps growing since its release in 2008 and has been dominating the mobile system market for a long time. According to the latest market statistics done by IDC, Android managed to capture 85.0% of the worldwide smartphone market share by the 1st quarter of 2017 (IDC, 2017). In the meantime, the global shipment of new Android devices is experiencing an average of 10% growth each year since 2015 (Linda, 2016). Due to people's heavy reliance on mobile devices and the popularity of Android mobile systems, the privacy concern and security issues on Android systems catch great attention from mobile users, industry players and academic researchers. At the same time, it also makes Android the prominent target of attackers. Unfortunately, Android vulnerabilities keep emerging and have successfully been turned into their exploitation even though Android has strengthened its security mechanisms and fixed a great number of vulnerabilities as its version evolves.

Vulnerability exploitation is a common way to achieve higher privilege on Android systems. Exploiting Android devices has been a popular topic since Android was firstly introduced in 2008. There are numerous exploits being implemented in the

* *Corresponding author.*
E-mail addresses: menghs@i2r.a-star.edu.sg (H. Meng), vriz@i2r.a-star.edu.sg (V.L.L. Thing), cheng_yao@i2r.a-star.edu.sg (Y. Cheng), daiz@i2r.a-star.edu.sg (Z. Dai), zhang_li@i2r.a-star.edu.sg (L. Zhang).

https://doi.org/10.1016/j.cose.2018.02.019

0167-4048/© 2018 Elsevier Ltd. All rights reserved.

Android history. From the users' perspective, an exploit program can help them to bypass the security mechanism of their Android devices to achieve better control of their devices by obtaining a higher privilege, e.g., rooting their devices. On the other hand, the exploitation could also be misused to gain the control of victims' devices where the attacker can obtain financial profit from selling users' privacy (e.g., account information). We intend to provide a holistic and comprehensive understanding of the exploits that can be used to attain higher privileges in Android system. It would be helpful in terms of understanding how individual exploits work and how the trend of the exploits on Android would be.

In this paper, we are going to present a survey on all the publicly available Android exploits gathered on the Internet. We provide a taxonomy of the Android exploits and analyze the similarities/differences and strengths/weaknesses. We demonstrate the trend of Android exploits by analyzing the development of each exploit category. Furthermore, we evaluate a group of exploits on our test devices. In summary, our contribution could be summarized into three points:

1) To the best of our knowledge, this is the first complete and exhaustive survey on publicly available Android exploits. By analyzing each exploit, we filter out those exploits with the same way of working but different nicknames and finally distill 63 different exploits. By referring to our survey, a reader can easily find out the affected device models and Android versions of a publicly released exploit as well as the vulnerabilities behind it.
2) This paper conducts a comparative and in-depth analysis of existing real-world Android exploits for the first time. We propose a taxonomy and accordingly initiate a classification of these exploits. We also carry out a comparison among different types of the exploits. By analyzing similarities/differences and strengths/weaknesses of each type of exploits, we point out the evolution of exploitation throughout the history of Android and forecast the future trends of the exploitation on Android devices.
3) With a large volume of information of these exploits being collected, we select a group of exploits by matching their targeting devices and Android versions to our test devices. Then we conduct an experiment to validate those selected exploits. By observing the experimental results, we present our evaluation result and discuss our findings correspondingly.

In the following section, we will first introduce the background of Android security mechanism and typical Android privilege escalation. We then propose a taxonomy on Android exploitation considering various perspectives in Section 3. In Section 4, we present the list of exploits gathered from multiple online sources, followed by analysis based on our classification results. As an important part of this survey, we also use a number of devices to evaluate applicable exploits. Section 5 shows the evaluation outcome and presents the discussion based on our findings. After that, the paper is concluded in Section 6.

2. Background

2.1. *The architecture of Android*

Android is a mobile operating system built upon a Linux kernel. Fig. 1 shows the layered architecture of Android. The concise architecture of Android can be depicted into 4 layers, kernel layer, middleware layer, framework layer, and application layer. The Linux kernel is the bottom layer of the Android platform which provides the basic functionalities of operating systems such as kernel drivers, power management and file system. The layer above the kernel is called Android middleware layer, which contains essential elements of Android as a mobile platform (Rangwala et al., 2014). There are two parts in Android middleware layer, i.e., the native components and the Android runtime system. Within the native components, the Hardware Abstraction Layer (HAL) defines a standard interface to bridge the gap between hardware and software. Compared with the drivers located in the kernel layer, Android HAL holds most of the hardware vendor specific implementation, for example, the APIs of audio device and camera (Google, 2017a; Muthumani,

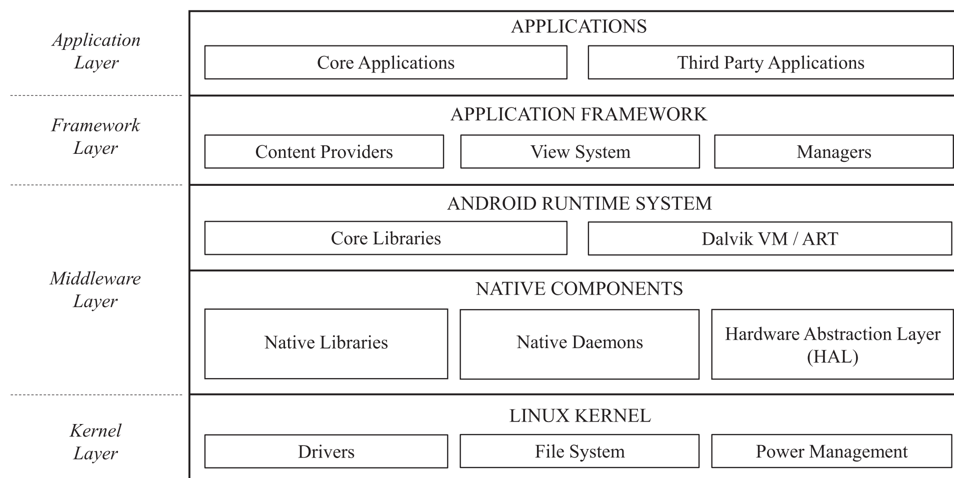


Fig. 1 – A layered architecture of Android operating system.

2015). The other two key components in the native components part are the native libraries and daemons which are written in C/C++. The native daemons handle all interaction with the system in native level. The native libraries, like SQLite, Webkit, SSL, and OpenGL, could greatly enrich the functionality and compatibility of Android platform for the development purpose. The Android runtime system contains the core libraries and runtime environment. A Java process virtual machine named *Dalvik* was used as the only runtime environment until the Android version 4.4. Thereafter, Android introduced a new runtime scheme called Android Runtime (ART) to replace the Dalvik virtual machine in later versions (Wikipedia, 2017a). Compared with the Just-In-Time (JIT) compilation used by Dalvik virtual machine, the Ahead-Of-Time (AOT) compilation provided by ART has been proven to have significant improvement in performance as well as energy consumption (Google, 2017b; Georgiev et al., 2014). On top of the Android runtime system is the application framework which is used most often by app developers as it handles many elementary functionalities of Android applications. For instance, the view system provides a rich and extensible collection of UI components; content providers enable an app to access or share data with other apps. All previously mentioned components build the foundation for application execution on Android platform.

2.2. *Security mechanism of Android*

There are two main security mechanisms on Android. One is the Android permission-based mechanism which is performed in Android application framework layer. The other one is Linux user-based privilege mechanism which is enforced in the kernel layer. An app must be granted with the corresponding permissions by the operating system prior to its access to the resources from the other parties (Google, 2017c).

The permission-based security mechanism is implemented at the Inter-Component Communication (ICC) level. As Android plays the role of reference monitor to mediate all ICC establishment, it regulates all ICC by assigning each application or component a pre-defined permission label. In this way, Android operating system will deny any ICC operation which asks for the permission beyond the pre-defined permission scope. Android introduces four permission levels for its access control mechanism. The lowest permission level is called *normal*. The permissions at this level can be granted as long as the developer declared them in the manifest file of an app, such as Internet access, vibration, and NFC usage. A higher permission level is named *dangerous* and the permissions at this level can only be granted after obtaining user's consent during the execution, for example accessing user's photos. The other two levels are *signature* and *signature and system*, which are designed for risky permissions. The former is only granted to those apps signed by a trusted party and the latter is granted by apps signed by Google and phone vendors (Google, 2017g).

Regarding the user-based security mechanism, each application on Android runs with a unique user identity, so that the underlying Linux system could provide the system-level isolation to refrain from damage caused by programming flaws (Bishop, 1996). However, there are some exceptions to system-defined privileged users, for example *root*, *system* and *radio* (Shabtai et al., 2010). A privileged user can initiate more than one process on Android system without the need to switch identity, and all of those processes are granted exact same privilege as the privileged user, which constitutes a potential security loophole in the Android system. Starting from Android system version 4.3, a Security-Enhanced Linux (SELinux) model has been enforced to upgrade the Discretionary Access Control (DAC) to the latest Mandatory Access Control (MAC). The access capability on Android has ceased to be solely determined by the file system ownership. Under the governance of SELinux, every process has to run at the minimum privilege level which is strictly regulated by a number of SELinux security policies (Shabtai et al., 2010; Google, 2017d).

2.3. *Privilege escalation*

Privileged access gives users the freedom to maximize the utilization of their Android devices. In Android, besides the "system" user, there is another pre-defined user called "root" which follows the "superuser" concept of Linux operating systems. The root user is provided with a full control of the system and furthermore, could access the user's data without any restriction (Bishop, 1996; Faden, 1999). In consideration of user privacy and system reliability, Google has neither produced any Android versions enabled with root permission nor encouraged people to root their devices since Android has been publicly released (Chris, 2012). Users will have to find a way to escalate their privileges if they want to achieve any functionality or customization which is not essentially granted by the Android system. Once users obtain root user privilege, they are able to back up apps and data in their preferred ways, recover files which are deleted by mistake, disable the advertisements or uninstall apps pre-loaded into the system image (OneClickRoot, 2017). According to a survey in 2014, over 27.44% of Android users rooted their device to uninstall useless and redundant built-in apps (Kristijan, 2014). From many users' point of view, rooting is a good resolution to their pain points during the use of the device. That may explain the reason why rooting has extremely high demands and is a popular topic in Android's world.

The process of rooting an Android device varies with versions of the operating system and hardware configuration. Rooting could be classified into two types depending on whether flashing the device is required or not (Zhang et al., 2015). The traditional rooting, which is so-called "hard root", attains root privilege by directly flashing superuser binary into the device. Hard root process comes with 3 steps: (1) unlocking the bootloader; (2) flashing a customized recovery image with superuser binary embedded; (3) installing a superuser permission management tool such as *SuperSU* (Yu, 2015; Martyn, 2016). The hard root method is simple but may lead to erasing all the data on the internal storage. The other type of rooting is called "soft root" or "indirect root". Soft root mostly refers to the scenario that the root privilege is obtained by running a software or process to exploit Android vulnerabilities. The soft root usually comes with a temporary privilege escalation on an active process. Nonetheless, it is also possible to make the rooting status permanent by making use of the current privileged process to copy the superuser binary to the system directory. Compared with hard root, soft root has a wider support on

different Android devices as it doesn't require a flash image for a specific device model and Android version. More importantly, soft root can root a device without any data loss.

Android rooting is a double-edged sword, the superior privilege obtained from rooting could not only provide users with more permission to use their devices or assist the government in the forensic investigation but also possibly expose all user's privacy and confidential information to the attacker (Zhang et al., 2015). Hence rooting could be a big threat to users' privacy and information security if it is conducted for the evil purpose.

2.4. *Vulnerabilities exploitation*

The history of vulnerability exploitation on Android device could be traced back to 2009, the second year when Android system was released. In that year, Christopher Lais implemented a utility called *Volez* to generate a crafted system recovery package. By making use of the code defect in over-the-air (OTA) recovery package verification on Android system, the Volez can add anything into the official package in the promise of validity. That exploit has been proved feasible on Motorola Droid when the first OTA update was published (Lais, 2009; Drake et al., 2014).

In 2010, Lucas Davi et al. presented an in-depth explanation of the exploitation with a component-based attack model (Davi et al., 2010). Suppose there is a non-privileged application (A_1) and a privileged application (A_3) running on a device simultaneously. The A_1 does not have the permission to access component of A_3. However, by making use of the conceptual weakness of permission mechanism of Android system, a non-privileged caller in A_1 could still have chance to access A_3 if there is an application in-the-middle (A_2), who allows the access from the unprivileged application A_1 and meanwhile has been granted the access to the privileged application A_3. Therefore the unprivileged application A_1 might always be possible to attain higher permission from another privileged application if the medium A_2 failed to implement necessary permission checks, as shown in Fig. 2. In a real-world application, any privileged software or service running on Android system can hold the role of A_2 in the model depicted by Lucas Davi et al. If there is a vulnerability being found in this privileged software or service, an exploit software, which plays the role of A_1, can be utilized by the adversary to corrupt the normal execution of A_2 and thereby obtain the superior privilege.

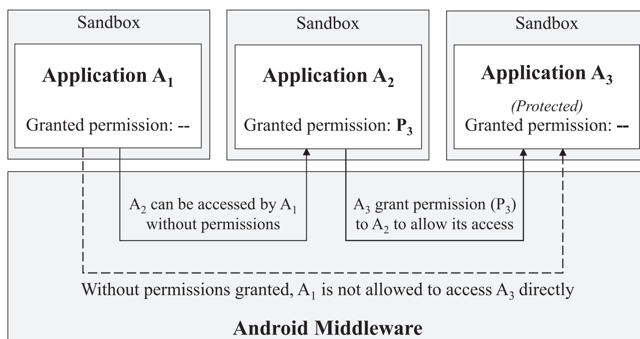


Fig. 2 – Component-based permission attack model.

In 2011, Höbarth and Mayrhofer discussed more possibilities to achieve privilege escalation by executing exploit program (Höbarth and Mayrhofer, 2011). They focused on the Android exploits that evolved from system level vulnerabilities and are initiated through native executable programs, and they categorized Android exploits into 4 typical attack methods, such as missing input sanitization, remapping shared memory, restriction of Anonymous Shared Memory (*ashmem*) access and overflow of process number.

Vulnerabilities may not only come from the privileged software made by Google but also possibly caused by defects from Linux Kernel, System On Chip (SoC) design, manufacturer's Read-Only Memory (ROM), carrier addition and privileged third-party applications. Once any vulnerability is found on the target device, it may give us an opportunity to exploit and thereby gain root permission (Google, 2014; Jon, 2014; Zhou et al., 2012).

The concern of exploitation based on the vulnerabilities has already been raised up in early years of Android history (Sadun, 2009). Starting from 2011, researchers started actively looking for security vulnerabilities on Android platform (Faruki et al., 2015; Felt et al., 2011; Vidas et al., 2011). Almost at the same time, various classification and survey of Android Common Vulnerabilities and Exposures (CVEs) were also conducted by researchers (Zhang et al., 2015; Drake et al., 2014; Xu et al., 2016). However, not all of the vulnerabilities can be exploited to escalate privilege and not all Android vulnerabilities come with exploits that are ready-to-use. It costs plenty of time and effort to understand vulnerability and come up with an exploit algorithm. As Android has greatly strengthened the security mechanism in recent years, the number of newly found exploitable vulnerabilities significantly reduced. However, new vulnerabilities keep emerging and have evolved to exploits successfully, such as *use-after-free* issue in Linux Kernel, Android keystore stack buffer issue and security weakness in *Android Trustzone* (Hay and Dayan, 2014; Shen, 2015; Xu and Fu, 2015).

3. Exploitation taxonomy

We propose a taxonomy for this survey to facilitate a holistic and comprehensive understanding of the Android exploits. With this taxonomy depicted in Fig. 3, we describe an exploit from 3 different perspectives – societal perspective, practical perspective, and technical perspective. From the societal aspect, we discuss who the potential attacker is, what is his or her motive in conducting the exploitation, and the possible consequence (risk) if such exploitation has been exercised. From the practical perspective, we discuss the pre-requisite for the exploitation, the steps to conduct exploitation, and the expected outcome (output) with the execution of the exploit. Finally, from the technical perspective, we discuss all the key elements of an exploit in an attack analysis model, including attack surfaces, attack vectors, and vulnerable targets.

3.1. *Societal perspectives*

3.1.1. *(S1) Attacker and (S2) Motive*

We use the term attacker to present the role of the entity who initiates the exploitation regardless if he or she is the owner

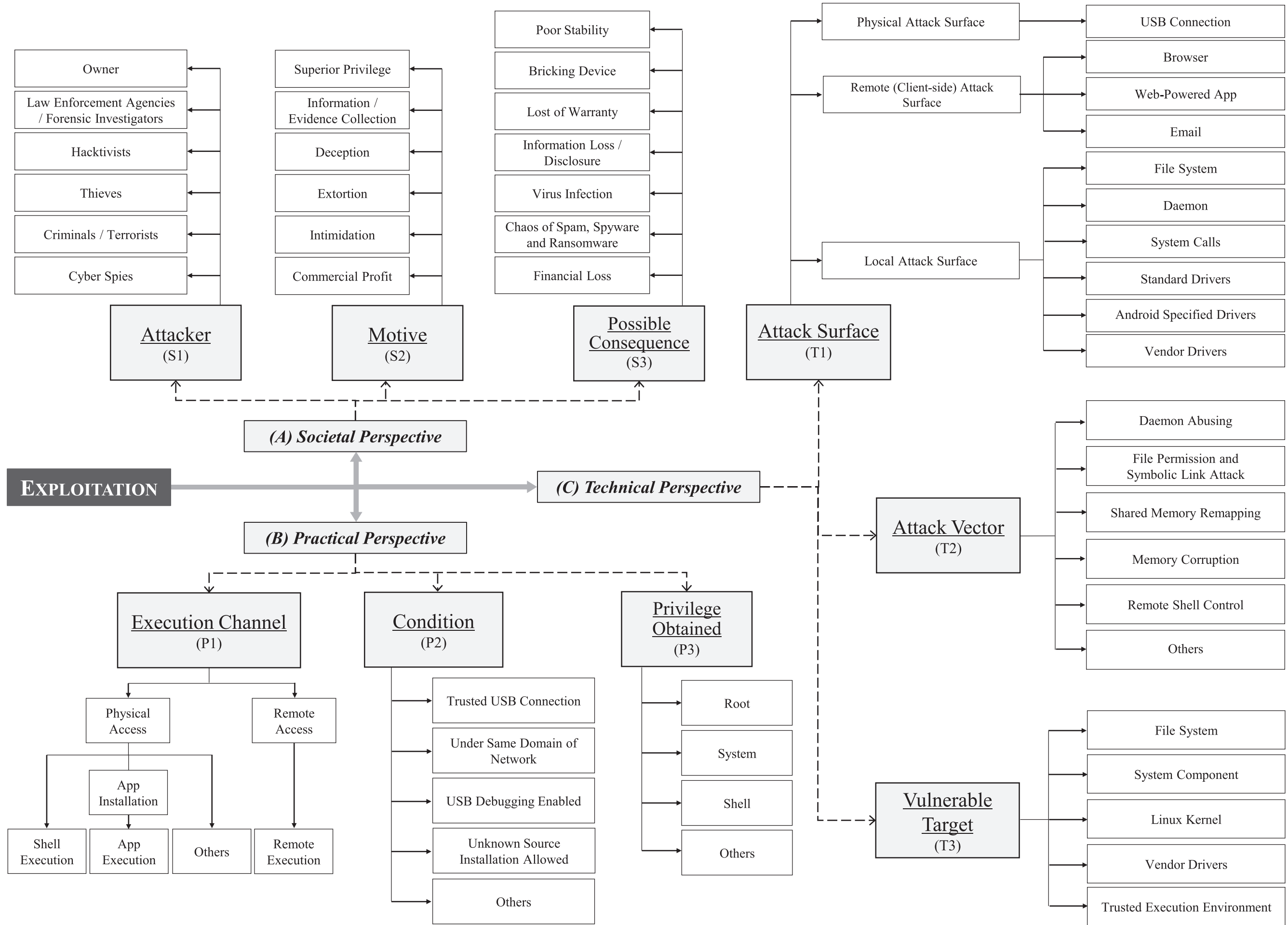


Fig. 3 – Android exploit attack taxonomy.

of the device or has malicious intention. In consideration of moral and legal qualms, most of the exploits are introduced to the public as a tool to enable devices' owners to gain superior privilege in their daily usage. However, it does not guarantee the exploit will never be used by people with malicious intent. Hacktivists and cyber-spies could profit from making use of exploit programs to gain control of the victims' device stealthily without users' consent or awareness. Thieves could continue using the stolen devices by obtaining superior privilege even if those devices have been locked by their owner. Terrorists can use exploits to control and interrupt the normal operation of people's smartphones. Moreover, exploit may also be used by the law enforcement agencies or investigators to obtain evidence for forensic purpose.

3.1.2. (S3) Possible consequence

Many exploits gain privilege with some side effects such as process crash or memory tampering, which may turn target device into an unstable status or even worse like being bricked resulting in loss of warranty. However, the risk of exploitation with malicious intention is much greater than the personal usage by the device owner. Attackers may inject virus or ransomware into the exploit program and install them in the target device once the exploit program acquired the superior privilege. The virus and ransomware could then reproduce themselves to harm other target devices. By stealing the sensitive information stored on the target device, the victim may possibly suffer from financial loss and leakage of commercial secret. The society may get into chaos and panics if the exploit has been utilized by terrorist to spread horrors and sabotage the communication functions on users' mobile devices. On the other hand, using exploits legitimately can bring greater possibility to the law enforcement agencies or investigators to find the key evidence from the devices of people involved.

3.2. Practical perspectives

3.2.1. (P1) Execution channel and (P2) Condition

Execution channel describes the approach that an attacker takes to the exploit program to conduct exploitation. Condition defines the pre-requisite that must be satisfied before the attack is exercised. An exploit may request more than one conditions to ensure a successful execution. The condition set of an exploit usually varies with the execution channel. With the knowledge of execution channel and all the necessary conditions of an exploit, we can construct a scenario to present the exploitation process on a real device. Here we define 4 different execution channels and we will discuss the conditions for each execution channel in following paragraphs.

App Execution: App execution describes the scenario that an attacker embeds the exploit code together with superuser binary files into an APK file and installs it on the target device to exercise the exploitation. The app containing exploit payload could be installed through on-device download or USB connection. Once the APK file has been installed on the target device, the privilege escalation can be triggered by any user's interaction within the app's interface, or even automatically while the app is running in the background. If the malicious code has been successfully executed, the privilege of the running process will be temporarily escalated, and then a persistent root could be obtained if the process with superior privilege copied the superuser binaries to the system executable directory within the Android system (Sun et al., 2015).

As the crafted APK containing exploit payload has almost zero chance to be published in Google Play market due to the pre-publish security check, the attacker must make sure the target device has enabled the "unknown source installation" option to allow the APK installation, and followed by the execution on the target device. Moreover, if the attacker wants to manually transfer the APK file to the target device through USB connection, the attacker also needs to ensure that the target device has enabled USB debugging and trusted the USB connection with the computer where the attack originates.

Shell Execution: Running executable scripts or binaries on the target device could be regarded as the most direct and effective way to conduct exploitation. To exploit, the attacker needs to connect the target device with the PC through USB connection, upload the script or executable binary along with all relevant files to a temporary directory on the target device by calling Android Debug Bridge (ADB for short) "push" command, and then execute the scripts or binaries in an ADB shell. Compared with the app execution, shell execution comes with a lower implementation cost and higher flexibility to apply to different operating system versions or device models. In addition, a successful shell execution usually results in a shell window with superior privilege, which brings convenience and freedom to the attacker to manipulate the target device.

Despite the strength that shell execution exploits have, there are still some restrictions that we should not ignore while conducting binary exploitation. Before the exploitation take place, what the attacker needs includes (1) a PC with drivers for both the target device and ADB installed; (2) the physical ownership of the target device to connect the device to the PC through USB at the moment of exploitation; (3) the password to unlock the screen of the target device, if any, to grant authorization and enable USB debugging and trusted connection.

Remote Execution: Remote execution is another optional execution channel to conduct exploitation on Android devices. It does not require the physical connection or APK installation. Instead, remote execution usually targets some native components of Android system, such as Webkit or media playback library, to attain the privilege escalation in distance (Seacord, 2015; Ben, 2010). In practice, the attack source may be a piece of malicious code pre-loaded into a web page or a crafted media file. The exploitation will be triggered at the moment when the target device user starts viewing or previewing the crafted web page or media file. During the execution, the attack source code could achieve privilege escalation and then initiate a remote connection between the target device and attacker's machine, and in the end, pass the full control of the target device to the attacker.

Remote execution stands out from the other exploitation channels that require a physical connection, and it is one of the trends of future exploitation on Android devices owing to many advantages it has (Wei et al., 2017). Firstly, the remote execution has excellent camouflage and anonymity because it is possible to attain high privilege on Android devices without a physical connection and victim's awareness. Moreover, as the attack originates from the network rather than local files or applications, it will be difficult for security mechanism in Android system or third-party anti-virus applications to detect the exploitation by local scanning or static analysis. Nevertheless, remote exploits still have some restriction when applying the exploitation in the wild rather than the laboratory. Even though such exploitation is physical connection-free, the attacker still needs to ensure that his or her attacking device is connected to the same network with the target device and the IP address of the attacking device must be filled in the malicious payload code prior to the exploitation.

Others: Besides those 3 types of common execution channels, there is an exception in the history of Android exploitation called *Volez*. *Volez* is the first ever publicly released exploit that takes advantage of one or more vulnerabilities of the target device to obtain root privilege. The *Volez* program can modify the factory OTA recovery image and insert `su` binary into the image. After that, the attacker copies the crafted image to the device storage and triggers the device recovery. The device operating system will be reset to factory status but has `su` binary located in system executable directory, which means the user of the device could easily gain root privilege by installing a superuser management app or calling "su" command in the ADB shell. As the *Volez* does not follow any attack channel where we mentioned above, we classify it and any similar future exploits as "others".

3.2.2. (P3) Expected privilege

Root privilege is the ultimate goal of privilege escalation on Android platform, however, sometimes it does not need to be mandatory as some functions like camera and screenshot do not require root privilege to invoke. Gaining code execution in Android *System Server* or *Media Server* could also be considered as a successful exploitation under certain circumstance (Davi et al., 2010). Many exploits can help attacker gain root privilege directly. Nevertheless, there are some exploits targeting high privileges other than root, for example, the system user privilege, the shell privilege, privileges within subsystems of the target device like baseband or trusted executable environment (TEE), etc.

3.3. Technical perspectives

3.3.1. (T1) Attack surface

Attack surface represents a set of interactions and components where an exploit takes advantage and initiates the attack routine. By observing various exploits, we summarize attack surfaces used by these exploits, furthermore we categorize those attack surfaces into 3 groups – they are remote attack surfaces, local attack surfaces, and physical attack surfaces. The

remote attack surfaces and local attack surfaces represent different approaches that the attacker uses to interact with the target device. On the other hand, the local attack surfaces contain the components within the Android operating system where the exploit takes advantage to escalate privilege. It is worth mentioning that an exploit may have multiple attack surfaces from different categories to constitute a successful attack.

(T1.1) Physical Attack Surface: Establishing USB connection between the attack machine and the target device is the first step for many exploitations. USB is the primary wired interface for Android devices to interact with other devices. In an active Android operating system, there is a service named ADB daemon (adbd) standing by all the time to facilitate the command operation and data transmission through ADB channel. Once a trusted ADB connection has been set up, the attacker can deploy an exploit by either executing corresponding commands to install a crafted application, or starting a shell session to run an executive file with attack payload. For that reason, USB is treated as the most ubiquitous physical attack surface exposed to diverse exploits on Android platform. In addition to the USB, there are some other physical connection methods applicable for some Android devices and theoretically feasible to be the attack surface of exploitation, for example, the SD Card and the HDMI connection. However, until the date of this paper being drafted, there is no publicly released exploit found to use any physical connection method other than USB.

(T1.2) Remote (Client-side) Attack Surface: The remote attack is always a very popular and attractive topic because it gets rid of physical restriction. Rather than a physical connection, the attacker could execute the exploit program over a computer network.

The web browser application is one of the major attack surfaces in remote attacks. One possible attack could be exercised by Document Object Model (DOM) manipulation through JavaScript. The malicious script injected by the attacker could dynamically modify the structure and content of current web page once loaded by the browser. In fact, due to the rich functionality of the browser application, there are a lot of opportunities for the attacker to explore the local attack surfaces from it. Among the existing Android exploits we surveyed, *Webkit Use-After-Free* is a typical browser attack which inserts a script into the space that has just been freed, and hereby achieves privilege escalation. Then it creates a remote shell window to allow the attacker to remotely control the target device.

Besides the browser, many web-powered applications on Android platform also has very high possibilities to be chosen as the attack surface during remote attacks. Most of the applications that work based on Internet service are implemented by making use of standardized web service APIs and libraries, for example, the SSL/TLS authentication and the embedded browser engine called *WebViews*. The components using standard WebViews libraries and APIs will very possibly reveal the potential attack surfaces. For example, in 2015, an exploit called *StageFright* has been found to take advantage of media previewing library on some Android devices. It can gain control of the victim's device through a reverse shell by sending a crafted media file which contains malicious payload and then being previewed within victim's device. A web-powered mobile application is selected to be the attack surface during the demonstration of StageFright exploitation.

Electronic mail (E-mail) client application is another potential attack surface for the remote attack. As most of the mainstream E-mail service providers allow users to attach any type of file in their messages, the attacker could insert malicious script into a document or media file and send to the victim through an E-mail message, then deliver an attack to the browser or other vulnerable applications to the victim's device.

(T1.3) Local Attack Surface: Local attack surfaces represent the attack point initiated by a program or script which is already executing on the victim's device. For most of the exploit programs, the local attack surface is the first step and a necessity of actual vulnerability exploitation. Here we summarize 6 different common attack surfaces in this category.

(T1.3.1) File System: Due to the Unix lineage of the Android system, the file system is one of the most frequently mentioned attack surfaces to conduct the local attack. The file system defines ownership and permission for each file entry. Attackers will have a chance to exercise exploitation if there is no sufficient restriction being enforced for the file permission assignment. One typical file system attack is taking advantage of inadequate security implementation in init.rc file to exercise a symbolic link attack (Drake et al., 2014). By changing /data/local folder permission to make it writable by the user and shell group, the user can reboot the device and then set the ro.kernel.qemu value to 1 in local.prop file. As the result, the user of device obtains the root privilege. We will discuss the real application of the exploitation targeting Android file system in next technical perspective T3.

(T1.3.2) Daemons: ADB and Zygote are two daemons most prevalently exploited by attackers. When we initiate an ADB session on Android devices, the ADB daemon (adbd process) starts running as the root user and then drops its privilege to the shell user before it gets ready to be used. However, in old versions of Android up to 2.2, the ADB daemon does not implement an adequate check to the return value of the setuid call at the moment of dropping privilege from root to the shell user. Similarly, the Zygote daemon also gives root user privilege to all the processes forked from it, and then drops privilege once the user who forks process from it has been confirmed. Since both daemons are originated from root user, they become popular attack surfaces to exercise an exploitation for root access. We will explain more about this process later in daemon abusing attack vector.

(T1.3.3) System Calls: System calls constitute an important type of attack surface in the Android system. An attack can exercise by making use of the vulnerabilities in the Linux kernel of the Android system, invoking system calls with malicious data or arguments and then tampering the system kernel to attain higher privilege. For example, *TowelRoot/futex exploit* is one of the best-known exploits in early versions of the Android operating system. It takes advantage of code defects in relock and requeue functions defined in futex.c in the Linux kernel source code. When attackers running the exploit program, those vulnerable functions will be called with malicious arguments and in an inappropriate manner. As a result, the addr_limit value of the current thread has been modified and the user

privilege of the current thread has been escalated correspondingly. Another exploit libperf_event targets on the perf_swevent_init function in the Linux kernel. It passes a negative integer as an argument of perf_swevent_init function to crash the current thread and then achieves arbitrary code execution to attain privilege for the attacker.

(T1.3.4) Drivers: In the Android platform, drivers usually represent a bundle of libraries or modules that bridge the gap between user applications and a hardware or native system service. In fact, there are lots of drivers existing in the operating system of an actual Android device. We group these drivers into 3 types on the basis of their functionalities.

The first type of driver is called standard drivers (more of Linux side), which are ported from the Linux kernel. This type of drivers usually serves as enablers of the basic hardware, such as Bluetooth and audio. The attacker can migrate a driver vulnerability or issue residing in the Linux system to the Android platform. For example, there is an exploit called *dirtyCow* which is essentially found on Linux platform but later proved that works on the Android operating system, too. It makes use of race conditioning vulnerability in tty (controlling terminal) driver, turns the read-only mapping of a file to writable status and finally gains privilege.

Another type of driver is Android specified drivers, which are implemented to support the exclusive functionalities of the Android system on the basis of the Linux kernel. Many features of the Android system such as the Anonymous Shared Memory (ashmem), binder, logger, and power management, are all enabled by Android specific drivers. Drivers belonging to this category can be treated as attack surface to conduct exploitation. For instance, a famous exploit during the early stage of the Android history called *KillingInTheNameOf* and its variant *psneuter* exploit are all using ashmem driver as the local attack surface.

Last but not the least, the vendor drivers also play a key role in the local attack surface family. Due to the uneven quality of the drivers' implementation by diverse hardware vendors, the APIs offered by vendor drivers often bring defects and therefore are used by attackers to conduct exploitation. Based on our findings, the first exploit that uses vendor driver as the attack surface is named *levitator* and released in 2011 (Drake et al., 2014). It takes advantage of the defect in APIs offered by PowerVR SGX chipset driver to corrupt the kernel memory and gain privilege. From then on, the exploits using vendor drivers as the primary local attack surface emerged quickly and reserved a significant proportion among all exploits since 2013.

3.3.2. *(T2) Attack vector*

Attack vector represents the concrete attack method used by an exploit program to gain privilege. The pair of attack vector and attack surface depicts an attack model of exploitation. For some complex exploits, there may be more than one attack vectors existing during different phases of an attack. We summarize the attack vectors for different exploits according to the taxonomy of attack surface we made.

(T2.1) Daemon Abusing: As we mentioned previously, the ADB daemon (adbd) in Android system is designed to drop its privilege from the root to shell before itself being presented for user interaction. But unfortunately, the ADB daemon does not properly check the return value of setuid function in Android versions prior to 2.3, which leaves a chance of gaining root privilege by interrupting the privilege dropping process to attackers. In 2010, Sebastian Kraphmer found there is a threshold value specifying the maximum number of co-existing ADB processes that the Android system could normally accomplish the privilege dropping action (Drake et al., 2014). Once the threshold value is being reached, the newly initiated ADB processes are presented to the user with root privilege. This kind of attack was later used in Sebastian's exploit named *RageAgainstTheCage*. Thereafter, abusing ADB has been mentioned again in 2012 in *Z2 exploit*. By making use of security issues in ADB backup function on some Sony Xperia devices, the exploit program creates a race condition and injects a privileged symbolic link into the system – by doing so the read-only system properties could be tampered and eventually the ADB will run as root user.

Abusing Zygote process is another example of daemon abusing on Android device. Zygote serves as the father process in the Android system, where all Android applications are started by being forked from the Zygote process. However, this forking procedure has similar privilege assigning routine with ADB daemon. When a new process is forked by Zygote, the Zygote by default gives the newly generated process with root privilege and then drops its privilege if the new process is initiated by the local user. Some exploits, such as *Zimperlich/zygote jailbreak* and *Zysploit*, make use of the similar defects in Zygote that the privilege lowering stage can be bypassed once the number of processes under one specific application user ID (UID) has reached its maximum value. As the result, the upcoming processes forked by Zygote under same application UID will run as root user.

(T2.2) File Permission and Symbolic Link Attack: File permission and symbolic link (symlink) attacks have been used in many exploits in Android versions up to 4.1. During the Android booting up, an init function will be called to execute the commands listed in init.rc script. The initialization commands contain a couple of folder creation (mkdir) actions, changing permission (chmod) actions and changing owner (chown) actions. Attackers usually look for security issues from this initialization procedure, and then create a symlink within the space where they are going to set with higher privilege. By doing this, attackers are able to make protected directories user-writable and then they can overwrite the local.prop file and set ADB user to root.

As most of these exploits make use of security issues in the third-party customized system image rather than stock Android, the file permission and symlink attack exploits are overall brand specific or device model specific, for example, the *TacoRoot* for HTC models, *TwerkMyMoto* for Motorola models and *NachoRoot* for Asus models. This type of attack vectors has been mitigated since Android version 4.2 as Android added extra security semantics in the execution of init script to prevent the permission and file system from being exploited.

(T2.3) Shared Memory Remapping: As mentioned earlier, Android has its own shared memory subsystem called `ashmem`. Android offers a number of richer and simpler APIs to programmers for performance improvement by enabling developers to utilize shared memory wisely. Android system allows any user to create and map a region in shared memory, and then execute read and write actions in a very efficient manner. Unfortunately, the `ashmem` might also be used by the attacker to

map and tamper the protected contents. Taking the exploit *KillingInTheNameOf* as example, due to the fact that system does not properly restrict local user to access the system properties space in early versions of the Android operating system like 2.x, the attacker could remap the space where local.prop is located, and thereby change the ro.secure property to make the ADB process bypass the privilege dropping operation in future booting.

(T2.4) Memory Corruption: Memory corruption is the attack vector that is commonly adopted for exploiting kernel and driver vulnerabilities. It occupies larger proportion among all available attack vectors for exploitation on the Android platform, especially after carrying a large scale of bug fixing and greatly strengthening Android security mechanism of Google since past few years ago. Memory corruption is a broad concept that describes all the approaches to alter the normal execution of the target device by memory manipulation. Many typical memory corruption methods like stack overflow, integer overflow, dereference of the null pointer and format error, could be found in available Android exploits.

To conduct an attack, the attacker usually chooses a privileged process as the target and then carries out memory attack by all means to crash the target process. Once the memory manipulation has been successfully achieved, the attacker may have a chance to execute arbitrary code on behalf of the privileged process and thereby gain soft root on the target device. For example, the *Use-After-Free Remote Code Execution on Webkit* uses null point dereference as attack vector to exploit; *GingerBreak* crashes the vold daemon by feeding a negative integer to it to cause an integer overflow; and *zergRush* invokes a libsysutils.so function with the wrong number of arguments passing in, leading to a Return Oriented Programming (ROP) chain to obtain soft root.

As the memory corruption is possible to occur at the moment of every function invocation during the execution of Android native program, there is no lack of memory corruption in our exploit survey results. It is hard to prevent memory corruption completely when compared with other types of attack vectors.

(T2.5) Remote Shell Control: All the attack vectors we list in previous paragraphs are corresponding to local attack. In addition to local attack, the attacker also has to own an extra pair of attack vector and surface to conduct a remote attack. Setting up a remote shell control is the most commonly used attack vector to conduct a remote attack. A remote shell control is not initiated by the attacker but triggered by the user of the target device – usually in an unaware manner. To prepare a remote attack, the attacker needs to implement the payload code which is able to start an ADB shell through the reverse TCP or HTTP connection to the attacker's machine, and then embed the code into the exploit program. Thereafter the attacker just needs to wait until the payload is triggered, followed by a reverse shell launched by the target device.

Gaining remote shell control does not bring the attacker with a root superior privilege. However, it links a remote circumstance up with local attack methods and thereby makes exploitation without physical connection possible. In our survey, we can find the presence of remote shell control in all remote exploits, for example, the *Use-After-Free Remote Code Execution on Webkit* and *StageFright*.

(T2.6) Others: Among the exploits we found during this survey, there are some outdated, unrepresentative or undisclosed attack vectors. For example, an exploit named *StumpRoot* doesn't provide any disclosure of technical details or source code to the public. The attack vector used by exploit *Volez* is outdated and unrepresentative when compared with other ones. In this paper, we classify those uncommon and unknown attack vectors as "others".

3.3.3. *(T3) Vulnerable target*

Vulnerable target describes the source of vulnerability where the attacker targets. An exploit could only be designed and implemented once the vulnerable target is confirmed. The mitigations of Android exploits are tailored to the vulnerable target correspondingly. In this paper, we present 5 different types of vulnerable targets to cover the existing cases of Android exploitation.

(T3.1) File System: The file system as a type of vulnerable target, is the most common focus of the file permission and symbolic link attack (refer to Section 3.3.2, Technical perspective T2.2). All Android exploits targeting file system are achieved by conducting file permission and symbolic link attack. The ultimate goal of an exploit targeting Android file system is to change the ownership of file for either content tampering or illegal execution. For example, changing the value of ro.kernel.qemu to 1 in the local.prop file.

Most of the vulnerabilities relating to this type of Android exploit are caused by the vendors' customization in Android, therefore the exploit attacking Android file system usually varies with devices' manufacturer and hardware configuration. The file system exploits used to be a common type of exploits on Android platform until the Android version 4.2 when the O_NOFOLLOW semantics has been introduced to prevent the symbolic link attack (Google, 2017e). Since version 4.3, Android enforces the SELinux model to regulate all permission involved activities within Android system.

(T3.2) System Component: We group all the Google-implemented components located in middleware layer, framework layer and application layer of Android architecture (refer to Fig. 1) as "system components". We use the term "system exploits" to represent exploits targeting vulnerabilities from system components. The Android components which provide either interface to native level development or user space access are often exploited by attackers. For example, the system services, native libraries, daemons, and Android shared memory (ashmem). The first ever publicly released exploit program on Android platform is a system exploit called "*Volez*", which manipulates the system recovery service to place superuser binary into the system executable directory on victim's device. There are some other milestones where exploits are categorized as system exploits, such as the *KillingInTheNameOf* that takes advantage of ashmem access issue to gain root access, the *RageAgainstTheCage* that targets vulnerabilities in Android *adbd* daemon to achieve privilege escalation; and the *StageFright* that can exploit the library with the same name in Android system to obtain high privilege in silence.

Compared with other 4 groups of vulnerable targets, the system components' vulnerabilities are relatively easier to be fixed by Android. The mitigation of system exploits usually

comes with a timely Android security patch or an operating system update.

(T3.3) Linux Kernel: We use Linux kernel to represent those vulnerable targets located in kernel layer components other than the file system, which includes Linux kernel drivers, process manager, and network controller. Unlike the Android system vulnerabilities, Linux kernel vulnerabilities are mostly found from the source code of the Linux kernel rather than Google's implementation in Android. Therefore, many vulnerable targets that exploit other operating systems in Linux family can also be migrated to Android platform. The exploitation targeting Linux kernel on Android platform has been firstly introduced in an exploit program named "*exploid*" in 2010. The *exploid* selects init daemon in Linux kernel as the target, exploits a vulnerability by initiating memory corruption and gains root privilege from the daemon.

Because the root user is designed as a part of privilege architecture in the Linux kernel, it is difficult for the components running on the Android platform to prevent privilege escalation initiated from the kernel exploits. Furthermore, due to the fact that the evolution of Linux kernel is usually slower than the evolution of Android, it makes it harder in Android to identify and fix kernel vulnerabilities within a short period. As a consequence, the Linux Kernel exploit usually works on multiple continuous version of Android systems.

(T3.4) Vendor Driver: The exploits in this category target hardware abstraction layer (HAL) implementation. The history of vendor driver exploits could be dated back to 2010. *Levitator* is the first Android exploit that conducts an attack on devices with specific hardware configuration containing PowerVR SGX chipset. As the Android experienced a rapid growth in the past few years, the great flexibility and diversity in hardware configuration on Android platform makes vendor driver exploits one of the major type of the exploitation.

Vendor driver exploits exclusively have 2 advantages. Firstly, compared with TEE exploits, vendor driver exploits can escalate privilege to "system" or "root" level and return to the user control directly; meanwhile, unlike the kernel exploits or system exploits, vendor driver exploits make use of vulnerabilities caused by the add-ons from various vendors rather than Android implementation, which means Android cannot exhaustively and effectively fix all the issues via its version update and hence may not be able to prevent from any induced exploitation in short time. Therefore, for the foreseeable future, we believe that the vendor driver exploits will still be the majority of new exploits.

(T3.5) Trusted Execution Environment: Attacking trusted execution environment (TEE) is a novel attempt at privilege escalation. In 2015, a researcher called "laginimaineb" published a series of blog articles and the proof-of-concept source code to explain his idea to gain TEE level privilege from *Qualcomm TrustZone*, which marks the first time that the TEE was exploited (laginimaineb, 2015).

Compared with exploits targeting other types of vulnerable targets, the TEE exploits have a very strict requirement but low practicability. Sometimes TEE exploits require a device flashed with customized ROM or that has Address Space Layout Randomization (ASLR) disabled. However, the outcome of a successful exploitation is just the privilege within TEEOS module rather than the root user. The attacker may not be able to attain full control of the victim device, instead, the attacker could gain access to the credential data like fingerprint or iris image, which is very sensitive as well. Currently TEE exploits are still in the theoretical stage, however, it is an inspiration to the future exploitation when security mechanism of the Android system becomes too strong to obtain root directly.

4. Survey and classification

We conduct a survey of publicly released Android exploits from multiple sources and we find 63 exploits covering all Android versions up to 7.0. By reading their descriptions, searching for available source codes and studying corresponding vulnerabilities, we collect rich details of these 63 exploits. In this paper, we summarize all the key details that are useful for upcoming analysis, and we organize them into a table. Table 1 shows the complete collection of all 63 exploits including their names, authors, release years, attack details, details of corresponding recorded vulnerabilities, affected devices and Android versions and our evaluation outcome. In order to distinguish an exploit from the others and facilitate our analysis, we focus on practical and technical perspectives of Android exploitation and we select 3 classification criteria from the taxonomy we proposed in the previous section. In the remaining contents of this section, we present our classification based on these 3 selected criteria and then we discuss our observation on each of them.[1] The classification is also included in Table 1.

4.1. *Execution channel*

We find all three kinds of execution channels from the 62 exploits except *Volez*. For convenience, we call those exploits using app execution channel as "app exploits", those exploits triggered by shell scripts through the physical connection as "shell exploits", and those exploits executed by embedded payload code via remote environment will be notated as "remote exploits". Fig. 4 demonstrates the distribution of exploits according to execution channels.

App Exploit. There are 19 exploits coming with a standalone APK file, reserving 30.6% among all exploits in our collection. All those exploits, except *GingerBreak* and *TowelRoot*, could only work on devices in a specific hardware configuration or in some exact models. The *GingerBreak* takes advantage of vulnerabilities in vold daemon and has been fixed in updates of Android version 2.3.4 and 3.0 in 2011, which are considered as a pretty early stage of Android history. Another exploit named *TowelRoot* used to be popular due to its long list of supporting device; however, its functionality to successfully gain root privilege stops at Android version 4.4.4, which is another outdated system nowadays (Wikipedia, 2017b). Therefore, we can optimistically conclude that most of the Android devices in the market are safe against existing APK exploits.

[1] As we mentioned in our taxonomy, we define those exploits with outdated, unrepresentative or undisclosed attack vectors as others. We do not consider exploits from this category in upcoming discussion due to their uncommonness.

Table 1 – Survey result of android exploits.

Year	Exploit	Author	Execution channel[1]	Attack vector[2]	Vulnerable target[3]	Vulnerability (CVE)	Vulnerability type (CWE)	Target devices	Affected versions	Verified
2009	1 Volez	Christopher Lais (Zinx)	O	O	SYS	N.A.	N.A.	Motorola Droid	2.0 and 2.0.1	–
2010	2 exploid	Sebastian Krahmer	S	M	KRN	CVE-2009-1185	Input Validation (CWE-20)	All	Up to 2.3.4	✓
	3 RageAgainstTheCage	Sebastian Krahmer	S	D	SYS	N.A.	N.A.	All	Up to 2.2	✓
	4 Use-After-Free Remote Code Execution on Webkit	Itzhak Avraham	R	R,M	SYS	CVE-2010-1807	Input Validation (CWE-20)	Unspecified[4]	2.0 to 2.1	–
	5 Zimperlich/zygote jailbreak	Sebastian Krahmer	S	D	SYS	N.A.	N.A.	All	Up to 2.2	✓
2011	6 KillingInTheNameOf	Sebastian Krahmer	S	A	SYS	CVE-2011-1149	Permissions, Privileges and Access Control (CWE-264)	All	2.1 to 2.2.2	✓
	7 psneuter ashmem exploit	Scott Walker (scotty2)	S	A	SYS	CVE-2011-1149	Permissions, Privileges and Access Control (CWE-264)	All	Unspecified	✓
	8 levitator	Jon Larimer and Jon Oberheide	S	M	VND	CVE-2011-1350; CVE-2011-1352	Information Leak/Disclosure (CWE-200) Buffer Errors (CWE-119)	Devices with the PowerVR SGX chipset	1.0 to 2.3.5	–
	9 Webkit use-after-free	MJ Keith	R	R,M	SYS	CVE-2010-1119	Resource Management Errors (CWE-399)	Unspecified	2.0 to 2.1.1	–
	10 Zysploit	Joshua Wise (jwise)	S	D	SYS	N.A.	N.A.	All	Up to 2.2	✓
	11 GingerBreak	Sebastian Krahmer	A	M	SYS	CVE-2011-1823	Numeric Errors (CWE-189)	All	2.1 to 2.3.3, 3.0	–
	12 sock_sendpage local root/ asroot/Wunderbar	Christopher Lais (Zinx)	S	M	KRN	CVE-2009-2692	Buffer Errors (CWE-119)	All	Up to 3.2.6	–
	13 zergRush	Revolutionary	S	M	SYS	CVE-2011-3874	Buffer Errors (CWE-119)	All	2.2.x till 2.2.2, 2.3.x till 2.3.6	–
	14 TacoRoot	Justin Case (jcase)	S	P	FLS	N.A.	N.A.	EVO 4G and some other HTC models	2.x	–
2012	15 NachoRoot	Justin Case (jcase)	S	P	FLS	N.A.	Permissions, Privileges and Access Control (CWE-264)	ASUS Transformer Prime	4.0 to 4.0.4	–
	16 TPSparkyRoot	sparkym3	S	P	FLS	N.A.	Permissions, Privileges and Access Control (CWE-264)	ASUS Transformer Prime	4.0 to 4.0.4	–
	17 mempodroid/mempodipper/ mem exploit	Jay Freeman (saurik)	S	P	KRN	CVE-2012-0056	Permissions, Privileges and Access Control (CWE-264)	Acer A200 Tablet, Galaxy Nexus, Motorola RAZR, Nexus S, Asus Transformer Prime	4.0.1 to 4.0.3	–
	18 Z2 Root Exploit	Sacha (xsacha), cubundcube and Andreas Makris (bin4ry)	S	D	SYS	N.A.	N.A.	Many Sony Xperia models, 3.x and 4.x before 4.1.1	Unspecified	–
	19 LG lit	giantpune	A	M	VND	CVE-2012-4220	N.A.	LG (with LM3530 backlight driver)	Unspecified	–
	20 Exynos Abuse/Sam (exynos-mem) Exploit	alephzain	S	M	VND	CVE-2012-6422	Permissions, Privileges and Access Control (CWE-264)	Samsung (Exynos 4 based)	Unspecified	✓
	21 diaggetroot	goroh kun	A	M	VND	CVE-2012-4220		HTC J Butterfly	Unspecified	–

Table 1 – Continued

Year	Exploit	Author	Execution channel[1]	Attack vector[2]	Vulnerable target[3]	Vulnerability (CVE)	Vulnerability type (CWE)	Target devices	Affected versions	Verified
2013	22 Qualcomm Gandalf camera driver exploit	alephzain	S	M	VND	CVE-2013-2595	Permissions, Privileges and Access Control (CWE-264)	Many models made by Asus, Huawei, LG etc.	Unspecified	✓
	23 Motochopper/ fb_mem_exploit	Dan Rosenberg (djrbliss)	S	M	KRN	CVE-2013-2596	Numeric Errors (CWE-189)	Motorola (Atrix HD, Razr HD, Razr M) and other devices with Snapdragon S4 series	Up to 4.2	✓
	24 libperf_event	Hiroyuki Ikezoe	S	M	KRN	CVE-2013-2094	Numeric Errors (CWE-189)	Nexus 4 and Some Japanese models made by HTC, Fujitsu, Sharp, Sony and LG	4.0 to 4.3.1 (Linux Kernel version before 3.8.9)	✓
	25 LG Sprite software backup/ LGPwn exploit	Justin Case (jcase)	A	P	FLS	CVE-2013-3685 (R)[5]	Race conditions	43 LG Optimus models	Unspecified	–
	26 libfj hdcp	fi01	S	M	VND	N.A.	N.A.	F05D, ISW11F and some other Docomo Fujitsu models	Unspecified	–
	27 Defy republic init_runit	Justin Case (jcase)	S	M	VND	CVE-2013-4777; CVE-2013-5933	Configuration (CWE-16) Buffer Errors (CWE-119)	Motorola Defy XT	2.3.7	–
	28 libdiag Exploit	Hiroyuki Ikezoe	S	M	VND	N.A.	Input Validation (CWE-20) Numeric Errors (CWE-189)	Many models made by NEC, Fujitsu, etc	Unspecified	–
	29 Boromir (camera-isp) exploit*	alephzain	A	M	VND	N.A.	N.A.	Many models (MTK based)	Unspecified	–
	30 Gemli (dev/DspBridge) exploit*	alephzain	A	M	VND	N.A.	N.A.	Many models (TI OMAP 36XX based)	Unspecified	–
	31 Frodo (exynos-mem) exploit*	alephzain	A	M	VND	N.A.	N.A.	Many models (Exynos based)	Unspecified	–
	32 Legolas (graphics/fb) exploit*	alephzain	A	M	VND	N.A.	N.A.	Many models (Exynos based)	Unspecified	–
	33 Aragorn (video1) exploit*	alephzain	A	M	VND	N.A.	N.A.	Many models (Exynos based)	Unspecified	✓
	34 Merry (s5p-smem) exploit*	alephzain	A	M	VND	N.A.	N.A.	Many models (Exynos based)	Unspecified	–
	35 Android put_user/get_user exploit *(Metasploit module)*	fi01, cubeundcube and timwr	R	R,M	KRN	CVE-2013-6282	Input Validation (CWE-20)	Unspecified	Linux kernel before 3.5.5 on the v6k and v7 ARM platforms, fixed in Jul 2013	✓
	36 TwerkMyMoto	Justin Case (jcase)	S	P	FLS	N.A.	N.A.	Motorola Razr I	4.1.2	–
	37 Pippin (memalloc) exploit*	alephzain	A	M	VND	N.A.	N.A.	Many K3V2 based models made by Huawei	Unspecified	–
	38 Gollum (amjpegdec) exploit*	alephzain	A	M	VND	N.A.	N.A.	Unspecified (AMLogic based)	Unspecified	–
	39 Faramir (camera-sysr) exploit*	alephzain	A	M	VND	N.A.	N.A.	Many models (MTK based)	Unspecified	–

Table 1 – Continued

Year	Exploit	Author	Execution channel[1]	Attack vector[2]	Vulnerable target[3]	Vulnerability (CVE)	Vulnerability type (CWE)	Target devices	Affected versions	Verified
2014	40 Barahir (Vcodec) exploit[*]	alephzain	A	M	VND	N.A.	N.A.	Many models (MTK based)	Unspecified	–
	41 WeakSauce	Justin Case (jcase) and Sean Beaupre (beaups)	A	P	FLS	CVE-2014-3847 (R)	N.A.	HTC One m7, m7 on Verizon, m8 and Droid DNA	4.1 to 4.4.4	–
	42 Qualcomm buffer overflow in acdb audio driver (msm_acdb_exploit)	fi01	S	M	VND	CVE-2013-2597	Buffer Errors (CWE-119)	Unspecified	Linux kernel 2.6.x - 3.x before Jun 2013	–
	43 Pie/vold asec	Justin Case (jcase)	S	P	SYS	N.A.	N.A.	Moto X on locked carrier	2.2.1 to 4.4.2	–
	44 TowelRoot/futex exploit	George Hotz (geohot)	A	M	KRN	CVE-2014-3153	Permissions, Privileges and Access Control (CWE-264)	All	Up to 4.4 ROMs built before Jun 2014	✓
	45 StumpRoot	IOMonster, Justin Case (jcase), autoprime and PlayfulGod	A	O	VND	N.A.	N.A.	Many models made by LG	Unspecified	–
	46 Android Browser exploit (ht_webkit_Android4)	Hacking Team	R	R,M	SYS	CVE-2011-1202; CVE-2012-2825; CVE-2012-2871	Input Validation (CWE-20)	Unspecified	4.0 to 4.3	–
2015	47 ObjectInputStream local root	Di Shen (retme7)	A	M	VND	CVE-2014-4322; CVE-2014-7911	N.A.	Nexus 5	4.4.4	✓
	48 libmsm memory corruption in camera driver (libmsm_vfe_read_exploit)	Hiroyuki Ikezoe	S	M	VND	CVE-2014-4321 (R); CVE-2014-4324 (R); CVE-2014-0975 (R); CVE-2014-0976 (R); CVE-2014-9409 (R)	Input Validation (CWE-20)	Unspecified	Unspecified	–
	49 PingPongRoot	Keen Team	A	M	KRN	CVE-2015-3636	Other (NVD-CWE-Other) Uninitialized data structure	Samsung Galaxy S6, Samsung Galaxy S6 Edge, HTC One (M9)	5.0 to 5.1.0	–
	50 Mate7 TrustZone Exploit	Di Shen (retme7)	S	M	TEE	CVE-2015-4421 (R); CVE-2015-4422 (R)	N.A.	Huawei Mate7	Unspecified	–
	51 Mtkfb Exploit	nforest@KeenTeam	S	M	VND	N.A.	N.A.	Unspecified (MTK MT658X and MT6592 based)	Unspecified	–
	52 Full TrustZone exploit for MSM8974	laginimaineb	S	M	TEE	N.A.	N.A.	Nexus 5	A crafted ROM built based on 4.4.4	–
	53 QSEECOM driver memory corruption	laginimaineb	S	M	TEE	CVE-2014-4322	Buffer Errors (CWE-119)	Unspecified	Unspecified	–
	54 StageFright Remote Code Execution *(Metasploit module)*	Joshua Drake (jduck) and NorthBit	R	R,M	SYS	CVE-2015-1538; CVE-2015-1539; CVE-2015-3824; CVE-2015-3826; CVE-2015-3827; CVE-2015-3828; CVE-2015-3829	Buffer Errors (CWE-119) Numeric Errors (CWE-189)	Nexus 5, Nexus 6, Nexus 7 and Samsung Galaxy S5 (SM-G900V)	5.0 to 5.1.1	✓

Table 1 – Continued

Year	Exploit	Author	Execution channel[1]	Attack vector[2]	Vulnerable target[3]	Vulnerability (CVE)	Vulnerability type (CWE)	Target devices	Affected versions	Verified
2016	55 mediaserver code-exec	laginimaineb	S	M	SYS	CVE-2014-7920 (R); CVE-2014-7921 (R)	Permissions, Privileges and Access Control (CWE-264)	Unspecified	4.3 to 5.1	–
	56 sensord local root	s0m3b0dy	S	M	VND	N.A.	N.A.	LG L7 and other devices have sensord deamon	Unspecified	–
	57 Metaphor	Hanan Beer@NorthBit	R	R,M	SYS	CVE-2015-3864	Numeric Errors (CWE-189)	Nexus 5	5.0 to 5.1.1	–
	58 iovyroot/pipe inatomic	idl3r	S	M	KRN	CVE-2015-1805	Code (CWE-17)	LG G Flex 2, many models made by Sony, Huawei and other brands	4.4.3 to 6.0	✓
	59 Use-After-Free camera driver exploit	betalphafai (Edward Hung)	S	M	VND	CVE-2015-0568	Use After Free (CWE-416)	Unspecified (Qualcomm MSM 7x30)	Unspecified (Linux kernel 3.0.x)	–
	60 QSEE TrustZone	laginimaineb	S	M	TEE	CVE-2015-6639	Permissions, Privileges and Access Control (CWE-264)	Nexus 6	Unspecified	–
	61 prctl_vma_exploit	betalphafai (Edward Hung)	S	M	KRN	CVE-2015-6640	Permissions, Privileges and Access Control (CWE-264)	Unspecified	5.1.1 and 6.0	✓
	62 Qualcomm TrustZone	laginimaineb	S	M	TEE	CVE-2016-2431	Permissions, Privileges and Access Control (CWE-264)	Unspecified	Unspecified	–
	63 Dirty Cow (dirtyc0w)	timwr	S	M	KRN	CVE-2016-5195	Race Condition (CWE-362)	Unspecified	Up to 7.0	–

[1]Legend for execution channels: (A) App Execution Channel; (S) Shell Execution Channel; (R) Remote Execution Channel and (O) notates Other Channels.

[2]Legend for attack vectors: (A) Shared Memory (`ashmem`) Remapping; (D) Daemon Abusing; (P) File Permission and Symbolic Link Attack; (M) for Memory Corruption; (R) Remote Shell Control and (O) Others.

[3]Legend for vulnerable targets: (FLS) File System; (KRN) Linux Kernel; (SYS) System, (TEE) Trusted Execution Environment and (VND) Vendor Drivers.

[4]The term “Unspecific” means the information has neither been disclosed by the author nor mentioned by any trusted source from Internet.

[5]The annotation “(R)” after a CVE identifier represents as “RESERVED”, which has been reserved for use by a CVE Numbering Authority (CNA) or researcher without all details being publicly disclosed.

*Discrete exploits integrated in *framaroot* root application. Those exploits could be initiated by either running a shell command or app execution.

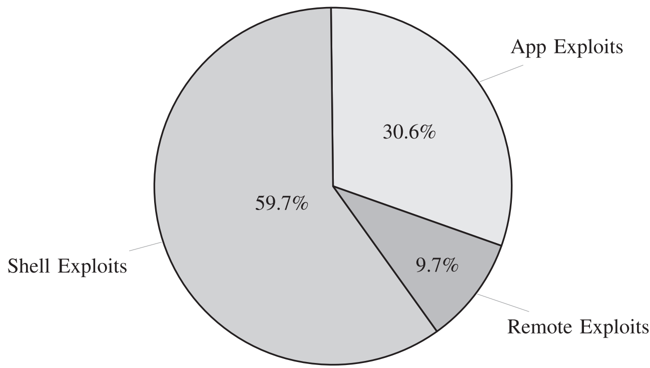


Fig. 4 – Type distribution of exploits by execution channel.

Shell Exploit. Shell exploits take the majority of our collection according to Fig. 4. We totally collect 37 shell exploits in this survey, which comprise over 59% of the entire collection. Among these 37 shell exploits, three exploits (*Defy republic init_runit*, *TwerkMyMoto* and *Pie/vold asec*) come with Java source codes that the attacker has to compile and compress them into an executable JAR file to conduct the exploitation; three exploits targeting the file system (*TacoRoot*, *NachoRoot* and *TPSparklyRoot*) are provided with an ADB shell script rather than executable binaries; and the 31 exploits left are all C codes which the attacker needs to configure the project with specified hardware architecture (e.g. arm64 or x86) (Google, 2017f) and build with Android Native Development Kit (NDK) to generate executable binaries. According to the information listed in Table 1, those 37 exploits cover all Android system versions up to 6.0 and most of the mainstream device models. Therefore in the case of all three pre-requisites (refer to Section 3.2.1, Practical perspective P2) of shell execution have been compromised to the attacker, an Android device will have a great probability to be successfully exploited if those shell exploits have been put into a proper combination and executed.

Remote Exploit. Other than local USB connection, ADB also provides developers with access to an unprivileged interaction through a remote shell. In our collection, 9.7% of exploits are classified as remote exploits. There are two exploits targeting the Android StageFright library vulnerabilities, three exploits targeting the Android Webkit library vulnerabilities, and one exploit attaining root privilege by taking advantage of a Linux kernel vulnerability. Taking *StageFright Remote Code Execution* (hereafter referred to as *StageFright* for short) as an example, it is a typical remote exploit that the attacker can gain root privilege of the target device by hosting a crafted web page containing media payload source code. The exploit will be triggered automatically and executed once the user starts browsing the crafted web page, and in the end, the exploit will initiate a remote shell process to the attack machine to pass the control of the target device to the attacker.

4.2. *Attack vector*

Classification by attack vector provides an intuitive perception to the methodology of Android exploitation. Based on the taxonomy proposed in the Section 3, we summarize attack vectors into 6 different categories. We demonstrate the growth of exploits in different attack modes throughout the history of Android system in Fig. 5.

Daemon Abusing Exploit. In this survey, we find four exploits that gain privilege by abusing daemons running on the Android system. Among the four daemon abusing exploits, the *RageAgainstTheCage* and *Z2 Root exploit* exhaust the ADB daemon (adbd) to interrupt the privilege dropping procedure of all future generated ADB sessions; the other two exploits – *Zimperlich/zygote jailbreak* and *Zysploit* impose similar routine to the Zygote daemon as the Zygote daemon also has privilege lowering mechanism while generating new process. The issues of these two daemons have been progressively fixed since the Android system version 2.3 was released at the end of 2010. According to our observation, the growth

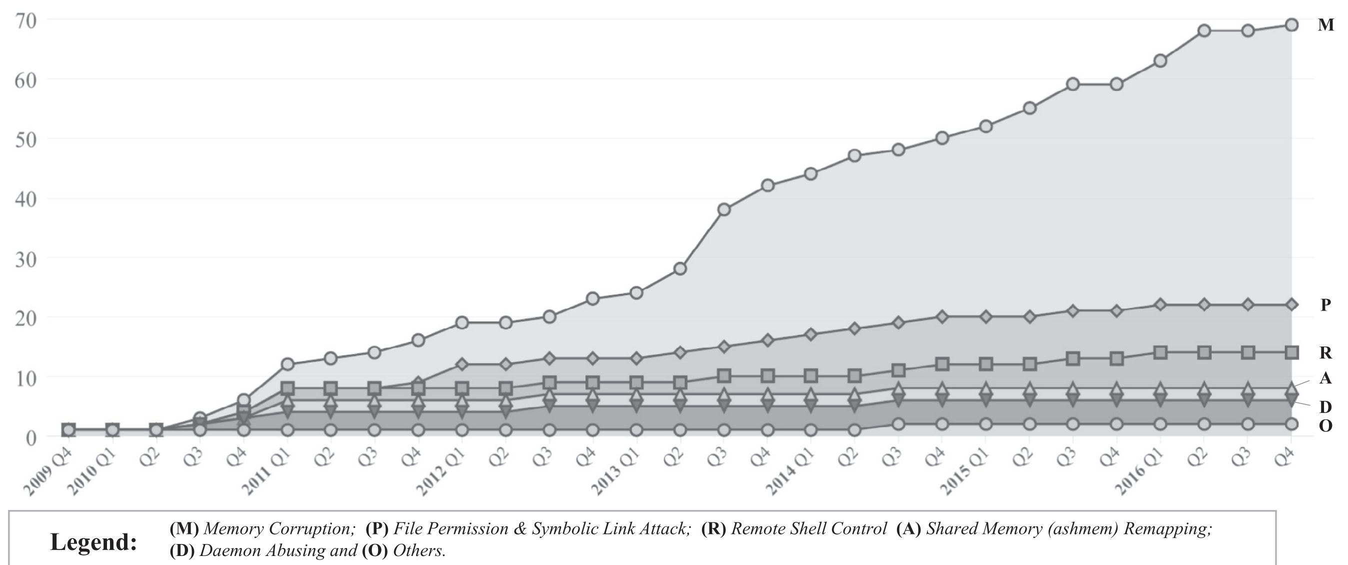


Fig. 5 – Growth of Android exploits with different categories of attack vectors from 2009 to 2016.

of this kind of attack has not appeared anymore in 2012 and afterward.

File Permission and Symbolic Link Exploit. File permission and symbolic link attacks are also known as *permission attack* or *symlink attack* elsewhere. There are 8 exploits gaining root privilege by initiating a permission attack. Except for the *mempodroid/mempodipper exploit* that makes use of a loophole from the Android file permission to directly modify process memory file, all the remaining 6 exploits take advantage of poor security mechanism in critical directories or files, create a symbolic link to either local.prop or uevent_helper and then tamper the privilege assignment of the ADB shell service. As a result, a successful exploit will return user an ADB session running as root user. The permission mechanism on Android file system is mature, safe and well organized. However, the attacker could still find a flaw to exercise attack due to the negligence or defect within the implementation of Android device manufacturer. Owing to the improvement of Android security mechanism and enhancement in security awareness from the device vendors, it is rare to see new file permission and symbolic link attack in recent years.

Shared Memory Exploit. Similar to daemons' abusing, shared memory remapping is another common approach to gain privilege in early versions of the Android system. We find two exploits that gain privilege by attacking the Android shared memory (ashmem) – *KillingInTheNameOf* and *psneuter ashmem exploit*. Both of them achieve privilege escalation by remapping the shared memory region and tampering the value of user ID assigned to ADB console. The ashmem vulnerability could be fixed by adding a number of authentication checks at the moment that the shared memory is accessed by any app or process from user space. In fact, the relevant ashmem vulnerability has been fixed in very short time. As Fig. 5 depicts, the number of shared memory attack cease to grow since 2011.

Memory Corruption Exploit. Memory corruption is the major attack vector throughout the history of the Android exploitation. Memory corruption exploits reserve over the half of all available exploits since the very beginning phase of the Android exploitation. Moreover, the memory corruption even becomes the only feasible local attack vector to gain privilege in 2015 and 2016. It is also worth noting that besides the quantitative growth, the diversity of memory corruption methods has been greatly enriched as well. During the early years, the memory corruption technique only covered format error (e.g. *exploid*) and null pointer dereference (e.g. *Use-After-Free Webkit*). But now, the memory corruption exploitation on the Android platform almost covers all the memory manipulation on conventional platforms, such as return-oriented programming. As it is impossible to completely prevent memory corruption in native level development for Android platform, we believe that memory corruption exploits will continue to play the primary role within Android exploit family in future.

Remote Shell Control Exploit. Compared with other attack modes, remote shell control usually requires the user of the target device, on purpose or unknowingly, to trigger the attack process. Conducting remote shell control attack could indirectly create an interface for the attacker and convert a remote attack to a local attack for further privilege escalation. For that reason, the remote shell control is usually executed together with a certain local attack vector to complete the privilege escalation operation for the attacker in distance. Remote shell control attack has a great advantage in camouflage and anonymity, which are two important features of a good exploitation. In this survey, we find 6 exploits that make use of this attack vector to achieve remote attack throughout the history of Android exploits. As mentioned earlier in Section 3.2 (Practical perspective P2), a successful remote shell control exploit requires the attacker to be under the same network with the victim's device and moreover to know the IP address of the victim's device prior to generating exploitation payload. Due to those reasons, even the growth of remote shell control exploits has never ceased since they were firstly introduced in 2010, the complicated execution procedure and strict conditions make the remote shell control still not yet a mainstream attack vector when compared with memory corruption.

4.3. *Vulnerable target*

Classification by vulnerable target could tell us where the exploitation originates. Moreover, we can gain an insight of the Android exploits in defense perspective by grouping the exploits by their vulnerable targets and analyzing the quantitative trend throughout the history of the Android system. According to the taxonomy in the previous section, we differentiate the vulnerable targets of all exploits into five groups of components within software part of the Android device. The classification result is also in Table 1 and we depict the quantitative trends of exploits with different vulnerable targets in Fig. 6.

File System Exploit. We find six cases of file system exploitation among 63 Android exploits, and all six file system exploits use file permission and symbolic link attack to gain privilege. By analyzing each of those 6 exploits, we notice that the device manufacturers' implementation is the root cause of the exploitation. We find that all of them are applicable to a small range of device models under the same brand, and the places where exploitation takes place are either the initialization scripts or the privileged factory software (e.g. backup and restore app). The history of file system exploits starts in 2011 when an exploit for some specific HTC models named "*TacoRoot*" has been released. In early 2012, the author of *TacoRoot*, Justin Case, released a variant called *NachoRoot*, which uses similar attack but is designed for a different set of device models. Shortly after that, another similar exploit *TPSparkyRoot* also has been published within the same forum, targeting same devices as *NachoRoot*. The other three exploits, *LG Sprite software backup/LGPwn exploit*, *TwerkMyMoto* and *WeakSauce*, have subsequently been released between 2013 and 2014 by Justin Case. At the same time, both Google and device manufacturers have greatly improved the security mechanism of Android devices. As the result, new file system exploits have ceased to appear since 2015.

System Component Exploit. System implementation is the second largest exploit target. There are a total of 16 system exploits released in the Android history, reserving over 25% among all surveyed Android exploits. Except for *Volez* which targets system recovery service and is counted as a special case for the early Android versions, all the remaining 15 system exploits are targeting native libraries, daemons or ashmem. Two exploits, *KillingInTheNameOf* and *psneuert*, take advantage of

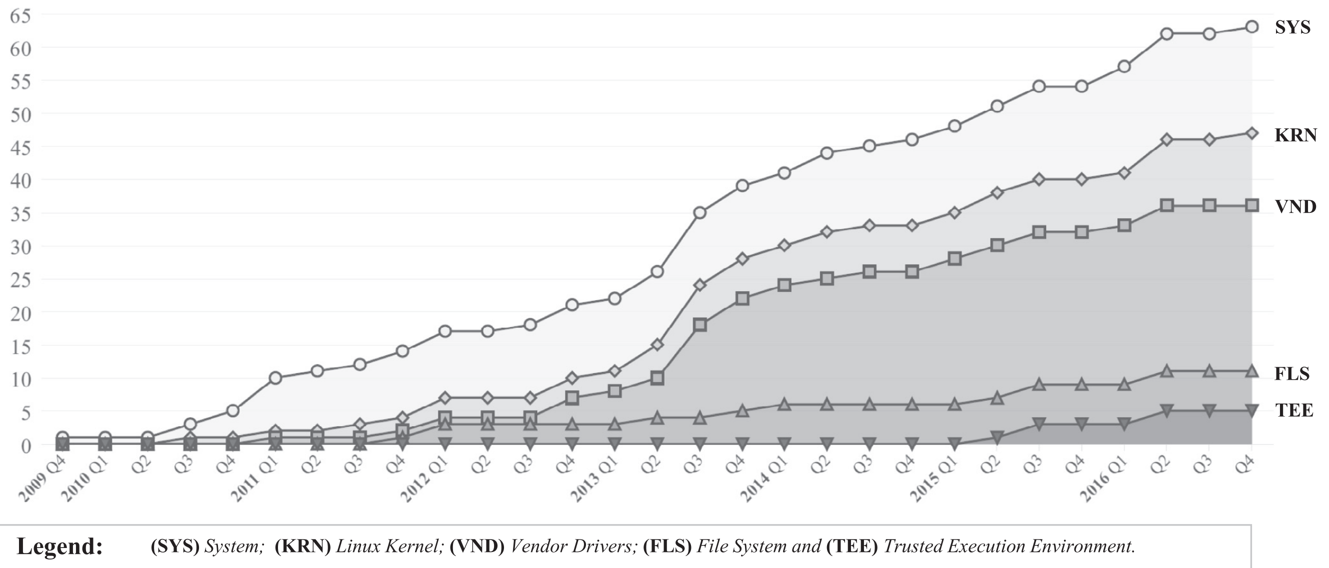


Fig. 6 – Growth of Android exploits with different categories of vulnerable targets from 2009 to 2016.

ashmem access issue and were published in 2011. Six exploits use daemons as their targets. The daemons which have been abused for exploitation include `vold`, `zygote` and `adbd`. Furthermore, there are seven exploits that conduct attacks by invoking system calls to specific libraries. The targeted vulnerable libraries include *Webkit*, *libsysutils*, *StageFright* and media server libraries.

It is noticeable to mention that the ashmem exploits only appear in the new exploit list of 2011 due to the quick fix. Considering the limited number of daemons that are accessible by user space processes, the growth of exploits caused by daemon abusing was suspended in 2014. Starting from 2015, all newly released system exploits are achieved by memory corruption against native libraries. There are a large number of libraries in the Android system implementation and each of them provides unique functionality and interface. Compared with ashmem exploits and daemon exploits which target only one or a few number of vulnerabilities, protecting the libraries from memory corruption attack is a more challenging work for Android.

Linux Kernel Exploit. Starting from the first Linux kernel exploit "*exploid*" which is released in 2010, the number of Linux kernel exploits (hereafter referred to as kernel exploits for short) keeps growing every year. In this survey, we collect 11 kernel exploits with their release date spanning from 2010 to 2016. From the static analysis made on those 11 exploits, we find there is only one kernel exploit called *prctl_vma_exploit* (2016) that is exclusive to the Android platform. All the other 10 kernel exploits are created as variants and share same vulnerabilities and attack routines with corresponding exploits for Linux operating system. Moreover, our analysis shows that kernel libraries and driver interfaces are the most frequently chosen vulnerable targets by Linux kernel exploits. Attackers often make use of flaws in input validation to create memory corruption while invoking kernel services. We also find that many Android exploits with long life cycle and wide support to different devices are kernel exploits. For example, the *TowelRoot* declares to be able to root all devices installed with Android version up to 4.4; and *PingPongRoot* could easily root over 100 models of latest device models at that time like Samsung S6/S6 Edge and HTC One.

Vendor Driver Exploit. There are 25 vendor driver exploits being included in our survey. We find that all vendor driver exploits are achieved by memory corruption except for the *StumpRoot* exploit (2013) which does not have enough details disclosed. In addition, we also find that vendor driver exploits usually come in a group to achieve the maximum effort and compatibility. For example, in 2012, an exploit making use of Samsung driver has been published by "alephzain". It is essentially named *Exynos Abuse* but later changed to *Sam exploit* when it is merged with some other vendor driver exploits written by the same author in 2013. The new bundle of various vendor driver exploits has been published as a one-click app on XDA forum with the name "*framaroot*". According to author's statement, the *framaroot* has integrated 12 different vendor driver exploits by 2014, covering over 450 device models from multiple brands (Alephzain, 2013).

Trusted Execution Environment Exploit. We find five TEE exploits during this survey. There are four of "laginimaineb" *Qualcomm TrustZone* exploits and 1 Huawei TEE exploit named *Mate7 TrustZone exploit* in our survey collection. By reading the authors' instructions of those exploits and analyzing their source code, we find that all of them can only achieve limited code execution with specific TEE level privilege, rather than flashing superuser binary or returning a root shell to the attacker. Therefore, in the current stage, TEE exploits still have obvious disadvantage in practicability and convenience when compared with others.

5. Evaluation and discussion

We perform an evaluation to observe the execution of exploits and validate their functionalities on Android devices. The

Table 2 – List of devices.

Device model and Code name	SoC model	OS
1 HTC Magic (HMA)	Qualcomm MSM7200A	1.6
2 HTC Hero (HHE)	Qualcomm MSM7200A	1.6
3 HTC One X *PJ46100* (H1X)	Nvidia Tegra 3 (AP33)	4.0.3
4 HTC One E9+ *0PJX100* (HE9)	MediaTek Helio X10 M (MT6795M)	5.0.2
5 HTC 10 *2PS6200* (H10)	Qualcomm Snapdragon 820 (MSM8996)	6.0.1
6a LG Nexus 4 *LG-E960* (NX4-4)	Qualcomm Snapdragon S4 Pro (APQ8064)	4.2.2
6b – – (NX4-5)	–	5.1.1
7a LG Nexus 5 *LG-D821* (NX5-4)	Qualcomm Snapdragon 800 (8974-AA)	4.4.4
7b – – (NX5-5)	–	5.0.1
8 LG Nexus 7 (2013) *LG-K008 (ME571K)* (NX7)	Qualcomm Snapdragon S4 Pro (APQ8064)	4.3
9 Samsung Nexus 10 *GT-P8110* (NX10)	Exynos 5 Dual 5250	4.2.2
10 Samsung Galaxy S2 *GT-I9100* (GS2)	Exynos 4 Dual 4210	4.0.3
11 Samsung Galaxy Note *GT-N7000* (GN1)	Exynos 4 Dual 4210	4.0.4
12 Samsung Galaxy S4 *GT-I9505* (GS4)	Qualcomm Snapdragon 600 (APQ8064AB)	4.2.2
13 Samsung Galaxy Note 3 *SM-N9005* (GN3)	Qualcomm Snapdragon 800 (8974-AA)	4.4.2
14 Samsung Galaxy S5 *SM-G900F* (GS5)	Qualcomm Snapdragon 801 (8974-AC)	5.0
15 Samsung Galaxy S6 *SM-G920I* (GS6)	Exynos 7 Octa 7420	5.1.1
16 Samsung Galaxy Note 5 *SM-N920I* (GN5)	Exynos 7 Octa 7420	5.1.1
17 Samsung Galaxy A8 Duos *SM-A800F* (GA8)	Exynos 5 Octa 5430	6.0.1
18 Samsung Galaxy S7 *SM-G930FD* (GS7)	Exynos 8 Octa 8890	6.0.1

testing has been conducted based on 18 different Android devices that we have. These 18 devices cover a wide range of manufacturers and system versions, including not only early and classical models in the Android history like HTC Hero, but also those later devices which are sold in smartphone market such as Samsung Galaxy S7. We filter out those exploits which are not compatible with our testing devices. Furthermore, we also remove an old exploit called *sock_sendpage* that we cannot manage to compile. As the result, there are 17 exploits being selected and formally tested in our evaluation. Those 17 exploits can be found in Table 1 where the check-marks are inserted within the column titled "verified".

5.1. *Preparation*

Before conducting the evaluation, we make sure every test device has Internet access and that both "USB debugging" and "unknown sources" options are enabled. Moreover, for the purpose of authenticity and accuracy, we also verify both root status and bootloader status of the target device to be negative before testing each exploit. We depict the 18 Android devices including their device models, hardware architectures and system versions in Table 2.

5.2. *Methodology and criteria*

There are two ways to gain root privilege among the 17 exploits chosen; the exploitation through physical access, and the exploitation through remote access. In this subsection, we describe the procedure of the exploitation imposed to our testing devices with respect to the two methods we mentioned.

1) *Exploitation through Physical Access:* According to the classification of exploits introduced in Section 4, both APK exploits and shell exploits are executed through USB connection between the attacker PC and the target device. All selected exploits except *StageFright* and *put_user/get_user* are imposed to the target device through the ADB channel.

TowelRoot is a typical APK exploit. In our evaluation, we install the APK by running "install" command through ADB and manually approve the installation on the target device. As *TowelRoot* is known as "one-click root solution", it is supposed to root any compatible device by just clicking the "root" button shown by the exploit app on screen. Besides *TowelRoot*, there is another APK called *Framaroot*, which is released by "alephzain", integrating a number of his exploits into one app. The installation of *Framaroot* is the same as *TowelRoot*. The only difference between *TowelRoot* and *Framaroot* is that the latter one offers a list of check-boxes to allow users to select which standalone exploit to execute. As the APK exploitation could not directly pass the gained privilege to the attacker, it usually copies a pre-loaded "su" binary to the system executable directory while the app has successfully obtained the privilege. To validate the outcome of exploitation, we can install superuser management apps such as *Root Checker*[2] or *SuperSU*[3] to check if the target device has been rooted or not.

For those shell exploits, we build the source code and export the executable binary in platform specified version that corresponds to the target device's hardware configuration, such as ARMv7 and ARMv8; then we push the executable binary to a temporary location in the target device system directory (generally /data/local/tmp) through ADB, open an ADB shell service, change the user mode of recently uploaded executable binary to make it executable by the shell user, and finally execute that exploit binary. In most cases, a successful shell exploitation will invoke the setuid call to escalate the current user to the "root" and in the end return to the same ADB terminal.

[2] Root Checker is identified as `com.joeykrim.rootcheck`.
[3] SuperSU is identified as `eu.chainfire.supersu`.

2) *Exploitation through Remote Access:* Compared with the prior solution, the remote exploitation could gain privilege of the target device in more casual and easier manner. There are two remote exploits being tested in our evaluation, *StageFright* and *put_user/get_user*. The evaluation of both exploits is conducted through *Metasploit*[4] which is a well-known penetration testing software. By making use of a designated *Metasploit* module, an app containing payload is generated, then the attacker, by all means, installs the app on the target device. After that, the attacker only needs to open the receiving port on the attack PC and keep listening in the Metasploit terminal. Once the app is launched on the target device, the payload code will automatically execute and initiate a reverse connection to the attacker. As a result, a shell process with superior privilege could be observed through a reverse TCP channel opened in the Metasploit interface on the attacker PC.

5.3. *Measurement and result*

The target devices presented in Table 1 are the theoretical prediction made based on the author's instruction, exploits' source code, and runtime behavior. And then we conduct the evaluation in accordance with the target devices summarized by us.

Taking *RageAgainstTheCage* as an example, we can find from the author's instruction that the exploit takes advantage of the vulnerability of Android system with version up to 2.3.4 and it does not vary with manufacture or any third-party configuration. Therefore we assume all Android devices loaded with systems with versions below 2.3.4 will be vulnerable to that exploit, and then we select devices that satisfy such requirement for evaluation.

However, there is another scenario that the author does not provide the detailed list of target devices of his/her exploit, like *libperf_hdcp* and *TowelRoot*. In that case, we do the static analysis on source code first, followed by a runtime analysis if the static analysis does not reveal any clue of target devices. In the source code of *libperf_hdcp*, we find a list of 25 constant strings looking like device model names. By searching each of them on the Internet, we finally confirm the list of target devices which includes *LG Nexus* 4 and a number of Japanese brand devices. Sometimes we encounter difficulty to conduct static analysis, for example, the *TowelRoot* as its source code is not provided. Based on the knowledge that the vulnerability used in that exploit only exists in Android system prior to version 4.4, we conduct an exhaustive testing on all our devices with system versions below 4.4. By observing the runtime logs, we find that besides the "succeed" and "fail", all devices which are not compatible with *TowelRoot* will print "try default" and followed by exit without any output regarding the exploitation outcome.

Finally, we note down the device models successfully exploited by the selected exploits as shown in Table 3.

After testing all 17 exploits on their designated devices, we have reproduced 15 out of 17 exploits on the actual devices. For the evaluation of the other 2 exploits, namely *Zysploit* and *prctl_vma_exploit* – we have not observed any positive result from executing them on all our devices that are supposed to be compatible. According to the result of the evaluation, there are 12 devices being successfully exploited for at least once in our experiment. The *LG Nexus 4* (4.2.2) compromises to 5 exploits, which marks the highest number among 18 devices. The runner-up in the ranking of successful exploitation is *LG Nexus* 5 (5.0.1), which has been exploited for 3 times. In contrast, some latest device models including *Samsung Galaxy A8 Duos*, *S7* and *Note* 5, are found to be immune from all the tested exploits.

[4] Metasploit is available at https://www.metasploit.com/.

Table 3 – Evaluation outcome.

Exploit ID and name	Result	Explanatory note
(2) exploid	✓	Devices: HHE
(3) RageAginstTheCage	✓	Devices: HMA
(6) KillingInTheNameOf	✓	Devices: HHE
(7) psneuter ashmem exploit	✓	Devices: HHE
(10) Zysploit	✘	No observation
(20) Exynos Abuse/Sam	✓	Devices: GS2, GN1
(22) Qualcomm Gandalf camera	✓	Devices: NX4-4
(23) Motochopper/fb_mem	✓	Devices: NX4-4
(24) libperf_event	✓	Devices: NX4-4, NX7
(33) Aragorn	✓	Devices: GS2, GN1
(35) Android put_user/get_user	✓	Devices: NX4-4, NX7, NX10
(44) TowelRoot/futex exploit	✓	Devices: NX4-4, NX7, GS4
(47) ObjectInputStream root	✓	Devices: NX5-4
(54) StageFright	✓	Devices: NX5-5
(58) iovyroot/pipe inatomic	✓	Devices: NX5-5
(61) prctl_vma_exploit	✘	No observation
(63) DirtyCow	✓	Devices: HE9, H10, NX5-5

5.4. *Discussion*

By doing this evaluation, we find that most of the exploits are able to gain privilege effectively on actual devices if all the requirements have been satisfied. At the same time, we also perceive that the real-world Android exploitation is not as simple and easy as what media describes in their articles.

Firstly, it is almost impossible to implement a new exploit that universally applies to all Android devices. The Android system has greatly improved its security mechanism during the past few years. Meanwhile, Google is also actively upgrading the Linux kernel along with the evolution of the Android system. As a result, those universal exploits like *RageAgainstTheRage* (up to Android version 2.2) and *KillingInTheNameOf* (up to Android version 2.2.2) have become history and we could seldom see new universal exploits being released nowadays.

Secondly, the high degree of hardware and software fragmentation in the Android ecosystem makes exploitation a challenging task. As more and more exploits use memory corruption technique to achieve privilege escalation, any slight difference in either Android version or hardware configuration may lead to variation of the address of a specific library in memory space, and thereby restricts the effect of exploitation. Not to mention the diversity in different manufacturers, the diversity in one device model family (e.g. *Samsung Galaxy S6*) already makes exploit difficult. Taking *Ping Pong Root* as an

example, it is known as a powerful root solution for Samsung Galaxy S6 family that claimed to support hundreds of different ROMs. However, we do not manage to find any ROM version compatible to the Galaxy S6 model in our lab. In the evaluation, we find that any incompatibility caused by inconsistency of Android versions and device models is very likely to make an exploit not work as it claims – this may explain the two negative results occurred during our evaluation.

Lastly, periodic security update mechanism, which has been adopted by more and more manufacturers, is transforming the traditional mitigation of exploitation. Nowadays protecting an Android device from being exploited could be done in a more effective and agile way rather than waiting for major version update of Android. Some manufactures like Google and Samsung actively push their regular security update or patch in a monthly manner to maximize the protection against the latest security threat. Compared with the major version update of Android, the security update may not easily be traced and analyzed. Therefore, from the perspective of exploitation, an attacker's exploit is very possible to be no longer effective if he/she exploits a vulnerability which is publicly disclosed. From the user's perspective, the risk of your Android device to be exploited could be significantly reduced if you enable periodic security patch update on your device.

Overall, our survey and experimental results unveil three trends of the Android exploits' evolution: (1) compared with the Android exploits released in early stage of Android history, the new Android exploits are more device specific and Android version specific – one exploit may only be compatible with one or a few device models with specific range of Android versions; (2) as the security mechanism of both the Android system and Linux kernel have been significantly strengthened, exploits targeting Linux kernel and Android system components experience decline; and vendors' customization becomes the prominent attack target in newly released exploits; and (3) due to the diversity of approaches and difficulty of absolute prevention, the memory corruption gradually becomes the primary attack vector to gain privilege on Android platform.

6. Conclusion

In this paper, we did a survey of publicly released Android exploits and proposed a taxonomy of Android exploits from multiple perspectives by analyzing the collected real-world exploits and conducting an evaluation of these exploits on a set of devices. We analyzed the characteristics of each category and presented the trend view of the Android exploits along the timeline from the technical perspective based on the exploit data. We also shared our discussion and outlook gained from the observation of evaluation.

REFERENCES

Alephzain. XDA Forums - [ROOT] Framaroot, a one-click apk to root some devices, 2013. Available from: https://forum.xda-developers.com/apps/framaroot/root-framaroot-one-click-apk-to-root-t2130276. [Accessed 14 March 2018].

Ben W. Researchers expose Android WebKit browser exploit, 2010. Available from: http://www.zdnet.com/article/researchers-expose-android-webkit-browser-exploit/. [Accessed 14 March 2018].

Bishop M. UNIX security: threats and solutions, 1996.

Chris H. The case against root: why Android devices don't come rooted, 2012. Available from: https://www.howtogeek.com/132115/the-case-against-root-why-android-devices-dont-come-rooted/. [Accessed 14 March 2018].

Davi L, Dmitrienko A, Sadeghi A-R, Winandy M. Privilege escalation attacks on Android, in: International Conference on Information Security. Springer, 2010, Conference Proceedings, pp. 346–360.

Drake JJ, Lanier Z, Mulliner C, Fora PO, Ridley SA, Wicherski G. Rooting your device. In: Android hacker's handbook. John Wiley & Sons; 2014. p. 73–81, [chapter 3].

Faden G. RBAC in UNIX administration, in: Proceedings of the fourth ACM workshop on Role-based access control. ACM, 1999, Conference Proceedings, pp. 95–101.

Faruki P, Bharmal A, Laxmi V, Ganmoor V, Gaur MS, Conti M, et al. Android security: a survey of issues, malware penetration, and defenses. IEEE Commun Surv Tutorials 2015;17(2):998–1022.

Felt AP, Finifter M, Chin E, Hanna S, Wagner D. A survey of mobile malware in the wild, in: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2011, Conference Proceedings, pp. 3–14.

Georgiev AB, Sillitti A, Succi G. Open source mobile virtual machines: an energy assessment of Dalvik vs. ART. In: OSS. 2014. p. 93–102.

Google. Google android security 2014 report, 2014, p. 7. Available from: https://source.android.com/security/reports/Google_Android_Security_2014_Report_Final.pdf. [Accessed 14 March 2018].

Google. Architecture – Android Open Source Project; 2017a. Available from: https://source.android.com/devices/architecture. [Accessed 14 March 2018].

Google. ART and Dalvik – Android Open Source Project; 2017b. Available from: https://source.android.com/devices/tech/dalvik/. [Accessed 14 March 2018].

Google. System and Kernel Security – Android Open Source Project; 2017c. Available from: https://source.android.com/security/overview/kernel-security.html. [Accessed 14 March 2018].

Google. SELinux concepts – Android Open Source Project; 2017d. Available from: https://source.android.com/security/selinux/concepts. [Accessed 14 March 2018].

Google. Security Enhancements in Android 4.2 — Android Open Source Project; 2017e. Available from: https://source.android.com/security/enhancements/enhancements42. [Accessed 14 March 2018].

Google. ABI Management – Android Developers; 2017f. Available from: https://developer.android.com/ndk/guides/abis.html. [Accessed 14 March 2018].

Google. Android Developers; 2017g. Available from: https://developer.android.com/guide/topics/manifest/permission-element.html. [Accessed 14 March 2018].

Hay R, Dayan A. Android keystore stack buffer overflow, 2014.

Höbarth S, Mayrhofer R. A framework for on-device privilege escalation exploit execution on Android, Proceedings of IWSSI/SPMU, 2011.

IDC. IDC: Smartphone OS Market Share, 2017. Available from: https://www.idc.com/promo/smartphone-market-share/os. [Accessed 14 March 2018].

Jon S. Practical Android exploitation, 2014. Available from: http://theroot.ninja/PAE.pdf. [Accessed 14 March 2018].

Kristijan L. Over 27.44% Users Root Their Phone(s) In Order To Remove Built-In Apps, Are You One Of Them? 2014. Available

from: https://www.androidheadlines.com/2014/11/50-users-root-phones-order-remove-built-apps-one.html. [Accessed 14 March 2018].
laginimaineb. Bits, Please! – Getting arbitrary code execution in TrustZone's kernel from any context, 2015. Available from: http://bits-please.blogspot.sg/2015/03/getting-arbitrary-code-execution-in.html. [Accessed 14 March 2018].
Lais C. Volez – Zen Thought, 2009. Available from: http://www.zenthought.org/content/project/volez. [Accessed 14 March 2018].
Linda S. Strategy Analytics: Android Captures Record 88 Percent Share of Global Smartphone Shipments in Q3 2016, 2016. Available from: https://www.strategyanalytics.com/strategy-analytics/news/strategy-analytics-press-releases/strategy-analytics-press-release/2016/11/02/strategy-analytics-android-captures-record-88-percent-share-of-global-smartphone-shipments-in-q3-2016. [Accessed 14 March 2018].
Martyn C. How to root Android phone, tablet, install custom ROM: beginner's guide, 2016. Available from: http://www.pcadvisor.co.uk/how-to/google-android/how-root-android-phone-tablet-unroot-summary-3342120/. [Accessed 14 March 2018].
Muthumani. Android HAL and Device driver architecture, 2015. Available from: https://www.e-consystems.com/blog/system-on-module-SOM/android-hal-and-device-driver-architecture/. [Accessed 14 March 2018].
OneClickRoot. Top 10 Root Apps for Android, 2017. Available from: https://www.oneclickroot.com/top-root-apps/. [Accessed 14 March 2018].
Rangwala M, Zhang P, Zou X, Li F. A taxonomy of privilege escalation attacks in Android applications. Int J Secur Netw 2014;9(1):40–55.
Sadun E. Android security vulnerability discovered – Ars Technica, 2009. Available from: https://arstechnica.com/information-technology/2009/02/android-security-vulnerability-discovered. [Accessed 14 March 2018].
Seacord RC. Mobile device security, in: Proceedings of the 3rd International Workshop on Mobile Development Lifecycle. ACM, 2015, Conference Proceedings, pp. 1–2.
Shabtai A, Fledel Y, Elovici Y. Securing Android-powered mobile devices using SELinux. IEEE Secur Priv 2010;8(3):36–44.
Shen D. Exploiting Trustzone on Android, Black Hat US, 2015.
Sun S-T, Cuadros A, Beznosov K. Android rooting: methods, detection, and evasion, in: Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices. ACM, 2015, Conference Proceedings, pp. 3–14.
Vidas T, Votipka D, Christin N. All Your Droid Are Belong to Us: A Survey of Current Android Attacks, in: WOOT, 2011, Conference Proceedings, pp. 81–90.
Wei F, Li Y, Roy S, Ou X, Zhou W. Deep ground truth analysis of current android malware, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2017, pp. 252–276.
Wikipedia. Android (operating system) — Wikipedia, The Free Encyclopedia; 2017a. Available from: https://en.wikipedia.org/w/index.php?title=Android_(operating_system)&oldid=794843196. [Accessed 14 March 2018].
Wikipedia. Android version history – Wikipedia, The Free Encyclopedia; 2017b. Available from: https://en.wikipedia.org/w/index.php?title=Android_version_history&oldid=781928647. [Accessed 14 March 2018].
Xu M, Song C, Ji Y, Shih M-W, Lu K, Zheng C, et al. Toward engineering a secure android ecosystem: a survey of existing techniques. ACM Comput Surv (CSUR) 2016;49(2):38.
Xu W, Fu Y. Own your android! Yet another universal root, in: WOOT, 2015, Conference Proceedings.
Yu KF. Rooting an android device, DTIC Document, Report, 2015.
Zhang H, She D, Qian Z. Android root and its providers: a double-edged sword, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015, Conference Proceedings, pp. 1093–1104.
Zhou Y, Wang Z, Zhou W, Jiang X. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets, in: NDSS, vol. 25, 2012, Conference Proceedings, pp. 50–52.

Huasong Meng received his Master of Computing degree in Infocomm Security at National University of Singapore in 2016 and B.Eng. (Hon.) degree in Computer Science at Nanyang Technological University in 2014. He is currently serving as a research engineer at Institute for Infocomm Research, A*STAR, Singapore. His working experience covers mobile security implementation for government, banking and financial industry. His research areas include mobile system security, vulnerability analysis and blockchain.

Dr. **Vrizlynn Thing** is the Head of Cyber Security & Intelligence Department at the Institute for Infocomm Research, A*STAR. She is also an Adjunct Associate Professor at the National University of Singapore, and holds the appointment of Honorary Assistant Superintendent of Police (SpecialistV) at the Singapore Police Force, Ministry of Home Affairs. During her career, she has taken on various roles to lead and conduct cyber security R&D that benefits our economy and society. She participates actively as the Lead Scientist of collaborative projects with industry partners and government agencies, and takes on advisory roles at the national and international level.

Yao Cheng received her Ph.D. degree in Computer Science and Technology from University of Chinese Academy of Sciences in 2015. She is currently a scientist at Institute for Infocomm Research, A*STAR, Singapore. Her research interests are in the information security area, focusing on vulnerability analysis, privacy leakage and protection, malicious application detection, and usable security solutions.

Zhongmin Dai leads the System Security Group of Cyber Security and Intelligence Department at the Institute for Infocomm Research (I2R), A*STAR, Singapore. He received his Bachelor of Computing and Master of Computing from National University of Singapore, in 2014 and 2017 respectively. His research interests include digital forensics, vulnerability analysis, cyber-security issues for autonomous vehicles and IoT.

Li Zhang received the B.Eng. (Hons.) and Ph.D. degrees from Nanyang Technological University (NTU), Singapore, in 2010 and 2015, respectively. He served as a security evaluator for smart cards in UL before joining the Cyber Security and Intelligence Department at the Institute for Infocomm Research (I2R), Agency for Science, Technology and Research (A*STAR) as a research scientist. His research interests include vulnerability detection, malware analysis and classification, and hardware security and trust.