

CS5250 - ADVANCED OPERATING SYSTEMS  
RESEARCH ASSIGNMENT

---

# Analyzing Android Rooting Methods From Operating System Perspective

---

By

NGUYEN QUOC BINH  
A0137763X



School of Computing  
NATIONAL UNIVERSITY OF SINGAPORE

APRIL 2017

## Contents

ABSTRACT .....	1
I. INTRODUCTION.....	1
I.1 Problem Statement and Related Work .....	1
I.2 Research Objective and Scope .....	1
II. ROOTING BACKGROUND .....	2
II.1 Platform Overview .....	2
II.1.1 Architecture of Android versus mainline Linux .....	2
II.1.2 Security enforcements.....	4
II.2 Background in Boot, Image, Recovery .....	5
II.2.1 Partition layout.....	5
II.2.2 Details in Bootloader .....	6
II.2.3 Boot sequence .....	7
II.2.4 Fastboot mode and Recovery mode .....	8
III. ROOTING UNDER OPERATING SYSTEM PERSPECTIVE .....	10
III.1 Rooting with unlocked bootloader .....	10
III.2 Rooting with locked bootloader .....	12
III.2.1 Case Study 1: Exploiting Race Condition (RageAgainstTheCage).....	12
III.2.2 Case Study 2: Exploiting the lack of message verification (Exploid) .....	14
III.2.3 Case Study 3: Exploiting the signed integer check (Gingerbreak) .....	15
IV. HARDENING THE OPERATING SYSTEM TO MITIGATE ROOTING.....	17
IV.1 Ensuring boot integrity with Verified boot .....	17
IV.2 Light-weight Address Space Layout Randomization with Retouching .....	17
IV.3 Multi-Level Privilege Escalation Prevention with ASF and ASM.....	18
V. CONCLUSION AND ACKNOWLEDGEMENT.....	20
VI. REFERENCES.....	20

## ABSTRACT

In recent years, mobile devices like smartphones, tablets, smartwatches have become a necessary part of our daily life. Those devices increasingly store private and sensitive data which makes them become attractive targets for malware and attackers. The vast number of those devices are using the Android Operating System. The core concept of security in Android is permission mechanism. However, recent studies show that this permission enforcement in Android is vulnerable to many types of rooting.

There have been lots of articles in security forums as well research papers mention rooting techniques. Those articles are rich but not comprehensible and systematic since they mostly indicate steps of rooting, but do not explain deeply about the underpinnings of the Operating System. This research aims to analyze the interactions inside the Android Operating under a rooting process, from power up until gaining full root privilege, to highlight the essences of its security design.

### Keywords

Android, Access permission, Privilege escalation, Rooting

## I. INTRODUCTION

### I.1 Problem Statement and Related Work

By default, the root access is blocked on stock Android devices for security reasons. If users would like to gain complete control over their Android devices with root permissions, they could do a rooting process. If the rooting process is successful, users can have full control of the kernel to install custom firmware, overclock the CPU, remove bloatware, etc.

The process of rooting Android Operating System has been investigated intensively in recent years. Many sophisticated rooting techniques have been developed and surveyed. A widely cited paper by (Enck, Ongtang et al. 2009) has a comprehensive topic on the access permissions inside the Android OS. (Drake, Lanier et al. 2014) presents detailed information about individually published exploits for gaining root. A survey by (Sun, Cuadros et al. 2015) performed manual and dynamic analysis on 182 selected Android applications to identify current rooting detection methods and evaluate their effectiveness. The papers of (Zhang, She et al. 2015), (Shao, Luo et al. 2014) also gives insightful discussions on rooting behaviors and methods to prevent rooting. A recent paper of (Shen, Evans et al. 2016) from Symantec Research Labs shows a detailed analysis on 6.14 million Android devices comparing rooted and non-rooted characteristics.

Most of those papers focus on discussing the problem under the viewpoints of Information Security and Privacy (such as: Malware analyzing, Anomaly detecting, Rule-based policy, etc.). However, the underpinnings of Operating System is not covered deeply. This brings me to the choice of my research topic.

### I.2 Research Objective and Scope

The goal of the research is analyzing the rooting methods to highlight the security design in low level (the kernel layer and middle layer) of the Android OS, compared with other mainline Linux distributions. The rooting and rooting mitigation methods are analyzed under the perspective of Operating System study, without considering details about distributions or device models. The research will spend a large part to discuss about the main components of the Android OS: boot sequence, boot images, recovery mode, etc. Basically, the rooting steps are simple but the background behind each step is complicated.

**The rest of the research is organized as follows:** The background of rooting is provided in Chapter 2. Chapter 3 analyzes various rooting methods under the viewpoint of Operating System study, and Chapter 4 discusses approaches to prevent rooting. The conclusion is in Chapter 5.

## II. ROOTING BACKGROUND

### II.1 Platform Overview

#### II.1.1 Architecture of Android versus mainline Linux

Android is a complete embedded software stack, comprising of a modified Linux-based kernel, middleware, and Applications. The architecture of Android contains three main layers:

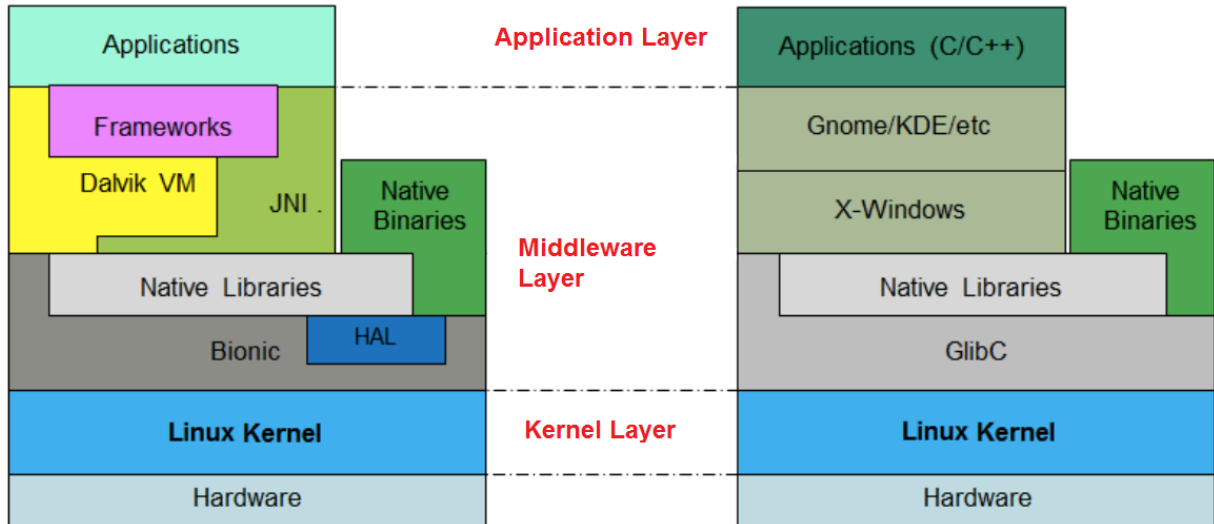


Figure 1. The architecture of Android compared with mainline Linux

#### Kernel Layer

Android relies on Linux kernel for core operating system services such as memory management, process management, network stack, security, etc. Device peripherals (GPS antenna, camera, accelerometer, etc.) are accessed through Linux drivers installed as kernel modules. Triggering peripheral drivers, as well as accessing the file system and memory is achieved by system calls.

The difference between the Android kernel and the common Linux kernel is Android kernel is customized specifically for the embedded environment consisting limited power, computing and memory resources. Android has modified the Linux kernel by adding components such as: Binder (Inter-Process Communication mechanism), Ashmem, Pmem, Low Memory Killer (for memory management), Logger, Alarm Framework and Wakelocks (for aggressive power management), RAM Console.

Kernel Changes	File	Descriptions
Anonymous Shared Memory (Ashmem)	mm/ashmem.c	A mechanism for anonymous shared memory, which abstracts shared memory as file descriptors. Applications can open a character device (/dev/ashmem) and create a memory region which can then be mapped into memory.
Pmem	drivers/misc/pmem.c	A mechanism for allocation of virtual memory that is physical contiguous. This is required for hardware which cannot support virtual memory, or vectored I/O devices that access multiple memory regions at once such as the camera.

Binder	binder.c	An IPC/RPC mechanism customized from OpenBinder protocol. Binder supports the communications between apps and system services by mediating the interaction between separate processes.
Wakelocks & Alarms	Kernel objects, exported to userspace via /sys/power files	Two power management extensions: Wakelocks enables Android to prevent system sleep and Alarms enables Applications to request a timed wake-up service.
Low Memory Killer	lowmemorykiller.c	A layer on top of Linux's Out-Of-Memory (OOM) killer, which terminates processes in case of memory exhaustion. While OOM killer is heuristic-driven, the Low Memory Killer in Android provides a more deterministic way of controlling process termination by setting memory pressure levels.
RAM Console	ram_console.c	Stores kernel log messages in RAM for viewing after a kernel panic, its content is accessible through /proc/last_kmsg
logger	logger.c	Android's light-weight and efficient logger, can be used by user-space components at run-time to log events regularly.

**Table 1. The typical changes in Android to Linux kernel**

### Middleware Layer

As can be seen from Figure 1, from this user-mode layer, the Android OS becomes more diverge from the standard Linux with these layers:

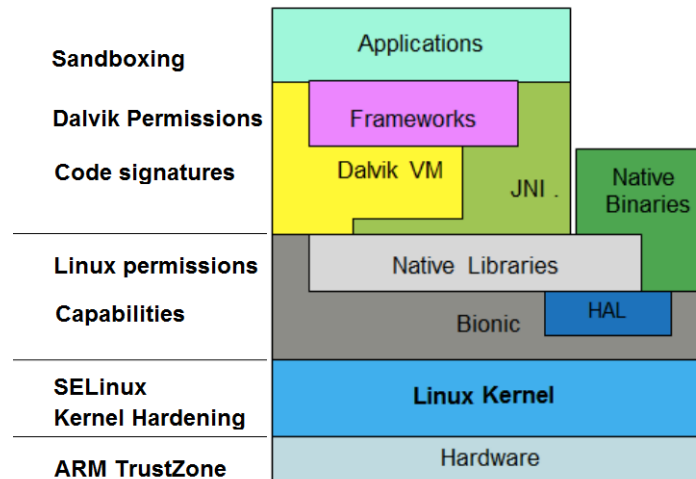
- **HAL (Hardware Abstraction Layer):** This layer aims to define an adapter to standardize different types of devices (phones, tablet, wearable devices, etc.).
- **Dalvik VM:** the Android runtime core component that executes Application files built in the Dalvik Executable format (.dex). The Dalvik VM is specifically optimized for efficient concurrent execution of virtual machines in a resource constrained environment.
- **Native C/C++ Libraries:** a set of C/C++ libraries that supports the direct invocation of basic kernel functionalities. The main part is Bionic libc, which is the own C-runtime library of Android.
- **Frameworks:** higher-level services of the platform, that are exposed to Applications as a set of Java APIs. Application developers can make use of these services.

### Application Layer

The layer comprises both user and system applications that have been installed and executed on the Android device. Each application comes with a set of permissions on the OS, that must be granted by the user upon installation to allow it to execute properly.

## II.1.2 Security enforcements

The security mechanisms are various and are implemented in all of three layers. However, they are revolved in the two primary concepts: permission and isolation (sandbox). By examining how permission and isolation are built, we will see how a rooting process can bypass those security mechanism to gain root access.



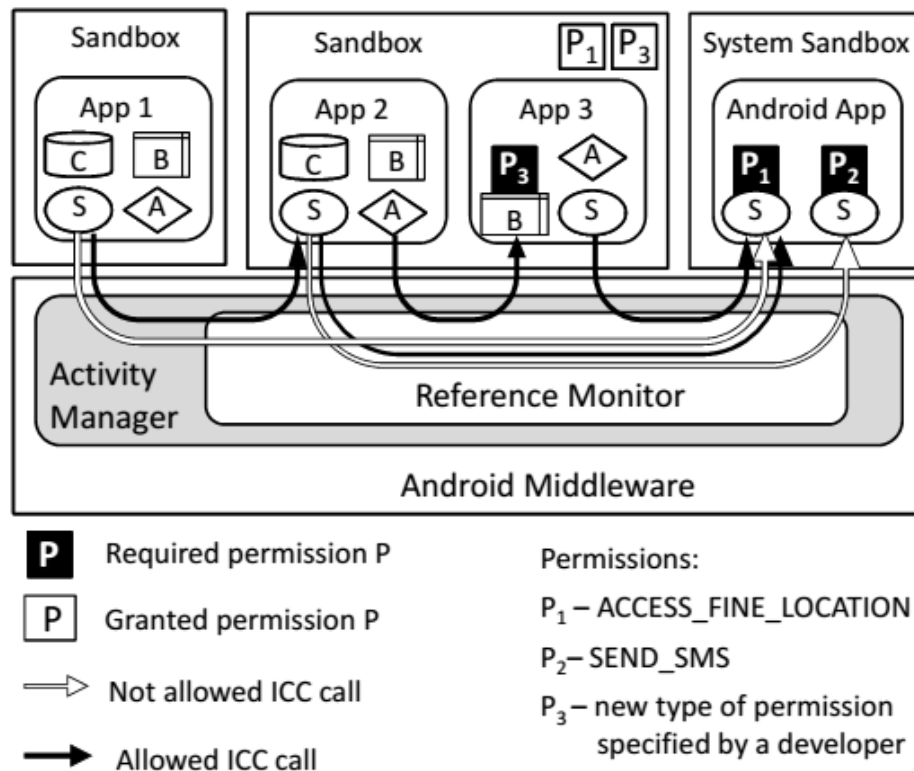
**Figure 2. Security enforcement in each layer of the Android OS**

**Permission mechanisms:** The Android permissions model are diverse. They can be classified into: API permissions, file system permissions, and IPC permissions.

- **API Permissions:** API permissions control the access to high-level functionalities within the framework layer. An example of a common API permission is `READ_PHONE_STATE`, which allows "read only access to phone state." An application that requests and is subsequently granted this permission would be able to call a variety of methods related to querying phone information.
- **File System Permissions:** The unique UID and GID of each application are given access only to its respective data storage paths on the file system by default. Subsequently, files created by applications will have appropriate file permissions set.
- **IPC Permissions:** Applying to the major application components that are built upon Android's Binder IPC mechanism. Each component can be either public or private. In case the component is public, other components can interact with it. In case the component is private, the only components that can interact with it are those that are part of the same application (or one that runs with the same UID).

**Application Sandbox:** Android inherits Linux's UID and GID paradigm, but does not have the traditional `passwd` and `group` files for its source of user and group credentials. Instead, Android defines a map of names to unique identifiers known as Android IDs (AIDs). The initial AID mapping contains reserved, static entries for privileged and system-critical users.

The goal of sandboxing is each Application is treated as an individual user from the OS's kernel view point. It ensures that one Application cannot modify (or even read, unless dedicated inter-process API calls are being made) the data of another Application.



**Figure 3. The sandboxing mechanism between each Android application.**  
Adapted from (Backes, Davi et al. 2014)

## Other noteworthy security enforcements in the Android kernel

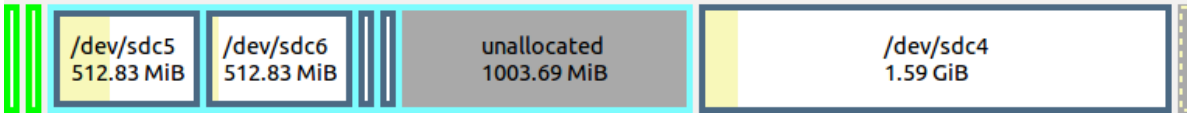
Some other security enforcements are: Hardware-based No eXecute (NX) to prevent code execution on the stack and heap, Address Space Layout Randomization (ASLR) to randomize key locations in memory. Also, there is full-disk encryption: using a single key, which is protected with the user password, to protect the whole userdata partition.

## II.2 Background in Boot, Image, Recovery

### II.2.1 Partition layout

There are typically seven partitions on Android devices: boot, recovery, system, cache, vendor, misc, user data. The rooting process will affect on the first three partitions.

- **Boot:** Containing the Android boot image, which consists of a Linux kernel (zImage) and the root file system ram disk (initrd).
- **Recovery:** Storing a minimal Android boot image that provides maintenance and recovery functions. By booting the device into the recovery mode, the normal boot process is circumvented, and the boot target is the booting currently loaded in this recovery partition.
- **System:** Storing the Android system image that is mounted as /system on the device. This image contains the Android framework, libraries, system binaries, and pre-installed applications.



Partition	File System	Label	Size	Used	Unused	Flags
/dev/sdc1	fat16	BOOT	21.98 MiB	5.44 MiB	16.54 MiB	boot
/dev/sdc2	fat16	RECOVER	21.98 MiB	5.65 MiB	16.33 MiB	
/dev/sdc3	extended		2.00 GiB	—	—	
/dev/sdc5	ext4	SYSTEM	512.83 MiB	189.45 MiB	323.38 MiB	
/dev/sdc6	ext4	CACHE	512.83 MiB	25.29 MiB	487.54 MiB	
/dev/sdc7	ext4	VENDOR	10.99 MiB	1.45 MiB	9.54 MiB	
/dev/sdc8	ext4	MISC	10.99 MiB	1.45 MiB	9.54 MiB	
unallocated	unallocated		1003.69 MiB	—	—	
/dev/sdc4	ext4	DATA	1.59 GiB	115.97 MiB	1.48 GiB	
unallocated	unallocated		2.30 MiB	—	—	

**Figure 4. The seven partitions on Android device**

- **Cache:** Storing frequently accessed and temporary data such as recovery logs and update packages. On devices with applications installed on SD card, it also contains the Dalvik VM cache.
- **Vendor:** Specific to vendors, used as a custom partitions for device configurations or upgrade operations.
- **Misc:** Containing different system settings, such as: CID (Carrier ID), USB configuration, and certain hardware information.
- **Userdata:** Storing user data and configuration. It also hosts private information of applications (such as login credentials).

## II.2.2 Details in Bootloader

Android vendors can implement their own Bootloaders freely. Most of them (including Samsung and Qualcomm) choose the "LK" (Little Kernel) Bootloader. LK is a minimal implementation of boot functionality. Essentially, LK is not a Linux kernel, but a bootable ARM binary image. LK consists of only the necessary functions of a bootloader, including:

- **Hardware initialization:** setting up vector table, MMU, cache, initialize peripherals, storage, USB, crypto, etc.
- **Finding and booting the kernel:** Locating the boot image and parsing its components (kernel image, ramdisk, and device tree), then transferring control to the kernel with a given command line.
- **Console support:** LK provides command interpreter and graphics functions, such as font support.
- **Supports flashing and recovery:** LK contains basic filesystem support, through lib/fs. Those functions help to enable the bootloader to erase or overwrite partitions during upgrade or recovery.

Bootloaders can be updated and flashed like other system images. The Bootloader image comprises of several sub-images, each of which is flashed to a particular partition. The image also contains the Resource Power Management (RPM) bootstrap, ARM TrustZone image, and secondary bootloader (SBL).



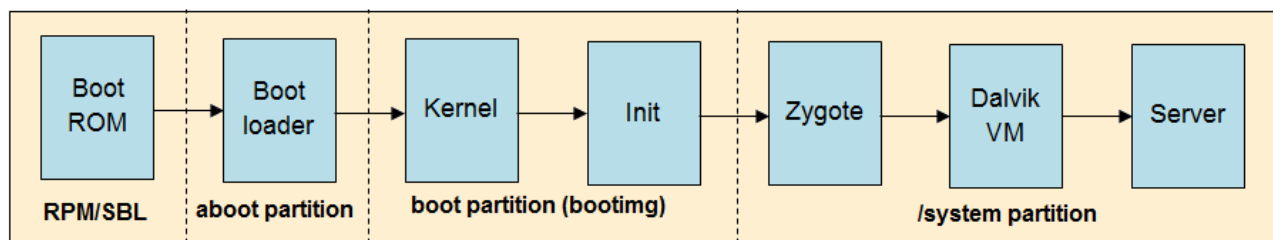
The classification of rooting methods revolves around Bootloader. Many Android devices are shipped with an unlocked bootloader, which allows device users to flash (write raw partition image to the persistent storage of device) custom system, recovery or boot images. Other Android devices do not have an unlockable bootloader. Locked bootloaders can load only boot images and recovery images signed by the manufacturer.

**Locked Bootloader:** this kind of Bootloader prevent the end users from modifying firmware by implementing many restrictions. Those restrictions can vary from manufacturers. The common method is using cryptographic signature verification that prevents booting from or flashing unsigned code to the device. The vendor provides its public key in ROM, and the key can be used to establish a chain of trust throughout the boot process so that all boot components can be verified. Those components contain a X.509 certificate, as well as the OpenSSL support needed to verify keys.

**Unlocked Bootloader:** this type allows device users to write raw custom system, recovery images or boot images to the persistent storage of the device. It also allows booting from transient boot environment without flashing them to the device.

### II.2.3 Boot sequence

The exact details of an Android boot sequence vary between devices and specific architectures. However, they can be generalized in the seven stages.



**Figure 5. Main stages in an Android Boot sequence with the corresponding partition**

**Stage 1 – Power up and executing Boot ROM code:** When power is stable, the execution will start with the Boot ROM code. The Boot ROM code starts to execute from pre-defined location, which is hardwired on ROM. It loads the Bootloader into RAM and starts the execution.

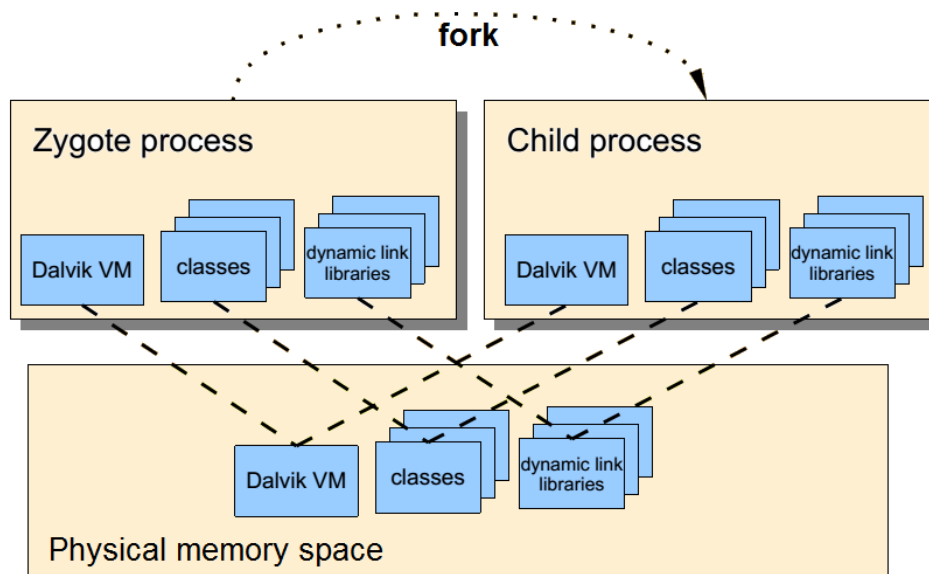
**Stage 2 – Bootloader:** The duty of a bootloader is to find and start the Android OS (normal boot mode) or recovery OS (recovery boot mode). First task is detecting and setting up external RAM. Once external RAM is available, the bootloader is loaded and placed in external RAM. Then the bootloader will look for a Linux kernel to boot. It will load the kernel from the boot media (or some other source depending on system configuration) and place the kernel in the RAM. It will also place some boot parameters in memory for the kernel to read when it starts up.

**Stage 3 – Kernel:** The Android kernel starts up in a similar way as on other mainline Linux systems. It initializes interrupt controllers; sets up memory protections, caches, and scheduling. Once the memory management units and caches have been initialized, the system will be able to use virtual memory and launch user space processes. The kernel will look in the root file system for the init process and launch it as the initial user space process.

**Stage 4 – Init:** The init process is the first process to start in the system by the kernel. The init process in Android will look for a file called init.rc. This is a script that describes the system services, file system and other parameters that need to be set up. Then, the init process will parse the init script and launch the system service processes.

**Stage 5 – Zygote:** Zygote process preloads typical classes and dynamic link libraries so that children processes can start quickly. It allows code sharing across the Dalvik VM.

**Stage 6 – Dalvik VM:** Android applications are packaged in Dalvik bytecode form and depend on the Dalvik Virtual Machine for interpretation at runtime. The Dalvik VM runs executable files in DEX (Dalvik Executable) format. The DEX format is designed for systems that are constrained regarding memory and processing power.

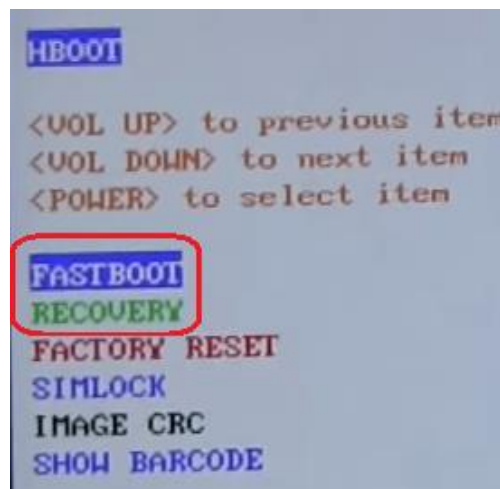


**Figure 6. Zygote and Dalvik VM**

**Stage 7 – Server:** The system server is the first Java component to run in the system. It will start all the Android services such as telephony manager, screen, etc.

## II.2.4 Fastboot mode and Recovery mode

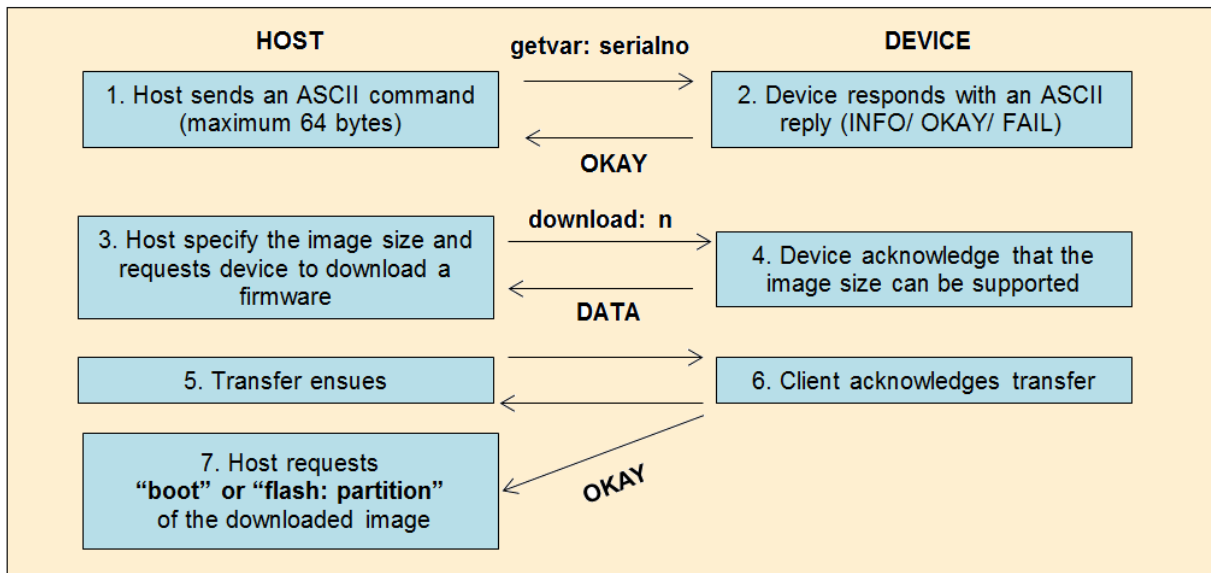
Beside the normal boot mode, each Android devices have two other different modes: Fastboot and Recovery.



**Figure 7. Recovery mode and Fastboot mode on the boot menu of HTC One**

**Fastboot:** It is the name of a tool and also a protocol for updating the flash file system in Android devices. After booting into fastboot mode and connecting the Android device to a computer via cable or mini USB, the device owner will have commands to re-flash the partitions, unlock the Bootloader, etc.

Some popular fastboot command are: “devices” used for listing devices, “reboot” for rebooting the device, “flash: partition” to flash image into a partition.



**Figure 8. The fastboot protocol**

**Recovery mode:** In this mode, instead of loading the standard boot image, the execution is diverted to the recovery boot image. The recovery boot image consists of three main components: header, kernel, and ramdisk.

```

66  /*
67  ** +-----+
68  ** | boot header      | 1 page
69  ** +-----+
70  ** | kernel          | n pages
71  ** +-----+
72  ** | ramdisk         | m pages
73  ** +-----+
74  ** | second stage    | o pages
75  ** +-----+
76  **
77  ** n = (kernel_size + page_size - 1) / page_size
78  ** m = (ramdisk_size + page_size - 1) / page_size
79  ** o = (second_size + page_size - 1) / page_size
80  **

```

**Figure 9. Contents of a boot/recovery image as defined in bootimg.h**

- **Header:** The header is placed at the start of the image and provides essential metadata to the bootloader, such as: a magic signature, the size and memory location in which to load the kernel, the physical address of kernel tags, command line options, checksum, etc.
- **Kernel:** The kernel file is based on the Linux kernel, with Android and device-specific modifications. It handles memory management, process management, network stack, driver, etc.
- **Ramdisk:** Containing the core files needed for system initialization, including a base directory structure, executable binary files, scripts launched by the device after booting the image, resources, and device configuration properties.

### III. ROOTING UNDER OPERATING SYSTEM PERSPECTIVE

The methods of rooting can be classified into two broad categories based on the status of the bootloader: rooting with unlocked bootloader and rooting with locked bootloader. In rooting with an unlocked bootloader, the approach is triggering the bootloader to boot into a previously modified boot region with available root access. In rooting with a locked bootloader, the root privilege is obtained by exploiting vulnerabilities in the Android OS.

#### III.1 Rooting with unlocked bootloader

In general Linux distributions, the task of switching from current user to root is done by the command `su` (“substitute user,” “super user,” or “switch user”). The `su` binary is used to allow a process to change the user account. Any process that requires root permission for their functionality must invoke `su`. In Android, only processes that belong to *root* or *shell* can become root. Hence, the ultimate goal of rooting with an unlocked bootloader is to install a customized `su` binary. That `su` binary must be executed by all applications and commands on the devices.

#### Step 1: Unlocking Bootloader

Most of Android devices have the bootloader locked by default. However, the major Android device vendors still provide official mechanisms that enable device owners to unlock bootloader, for example:

- Xiaomi provides Bootloader Unlock Guide at: <http://en.miui.com/unlock/>
- In their developer portal (<http://www.htcdev.com/bootloader/>), HTC states that they allow their bootloader to be unlocked for 2011 models onward.
- According to Sony Developer World (<https://developer.sonymobile.com/unlockbootloader/unlock-yourboot-loader/>), users can unlock the bootloader for a broad range of Sony devices.

Those mechanisms are slightly different from vendors. Concisely, they can be generalized as the below flow:

1. First, the device owner uses fastboot command (mentioned in Chapter II) to get the phone’s unlock token.
2. Then the unlock token is submitted to the Original Equipment Manufacturer (OEM) unlock portal.
3. The portal will validate the token and send back the unlock key.
4. Finally, the device owner uses the unlock key and fastboot command to unlock the device.

#### Step 2: Make a customized boot target with available `su` binary

A custom `su` binary is essential for gaining persistent root access. There are several ways to include the appropriate `su` binary in the system partition:

- **Replacing the system partition:** This approach is flashing a whole new system image, which contains a configured su binary, into the device.
- **Boot from a transient boot image:** The ramdisk in the Android boot image contains a root file system (rootfs) mounted as read-only at the boot time. Once that root file system is mounted, the configurations files (init.rc) will be executed. By making a boot image with customized scripts, we can boot the device from that image and then install the su binary into system partition.
- **Replacing the recovery OS by updating:** Applying an Over-The-Air (OTA) update package to change the recovery OS. OTA is the default Android update mechanism. An OTA update package is a .zip file that contains an executable update binary file. After the signature on the package is verified, recovery extracts this binary and runs it. Data is pulled from the update package and used to update the boot, system, vendor partitions.

### Step 3: Create a Superuser app to wrap the installed su binary

After having the su binary installed, we should create an application, usually called Superuser to “wrap” the su binary. There are many available Android app known as the name Superuser, and can be downloaded from different sources. The most famous among those Superuser apps are: Chainfire (<https://www.chainfire.eu/>) and Clockworkmode (<http://www.clockworkmod.com/>). Such apps are used to grant or deny other apps's requests for root access. Those apps operate under the root-privilege management model as below:

1. The app that needs root privilege first invokes the su binary. Then, su binary sends a root request via a messaging object to Superuser.
2. Superuser checks its policy database to decide whether it should grant the access or not. If there is no corresponding policy exists in the database, Superuser launches a popup window to ask the device user for the decision.
3. After getting the decision (granting or denying), Superuser sends it back to su binary via a local socket. Based on the message, su gives the requesting app the root privilege or denies it.

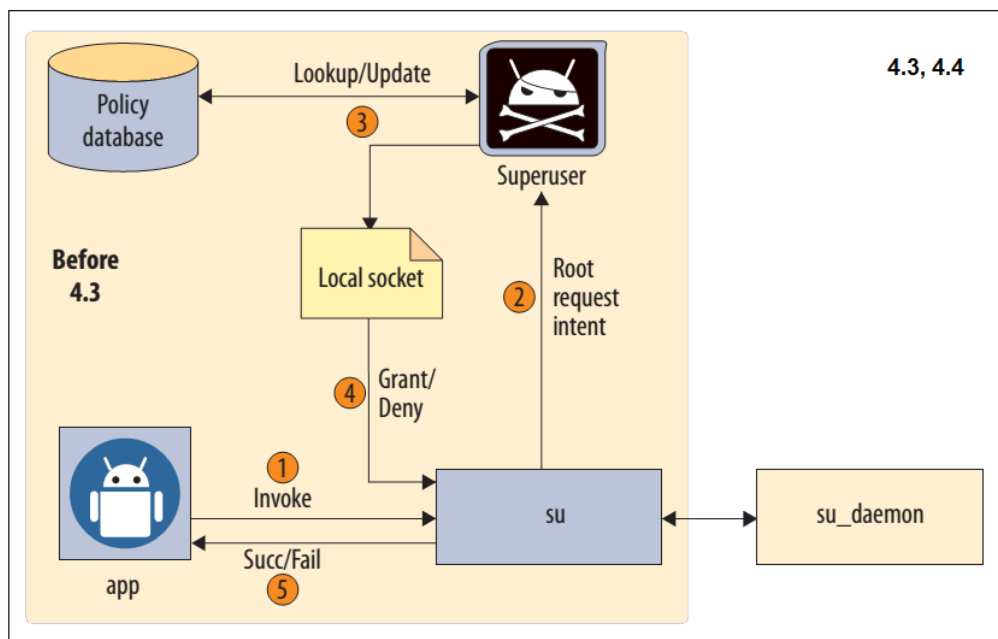


Figure 10. The root-privilege management model of Superuser

For pre-Android 4.3 versions, when an application invokes su, the root request will be sent to the Superuser application. For Android 4.3 and later versions, commands must be forwarded to su\_daemon, which is an additional security feature. Basically, su\_daemon is a root privilege daemon started during the boot sequence without capability.

## III.2 Rooting with locked bootloader

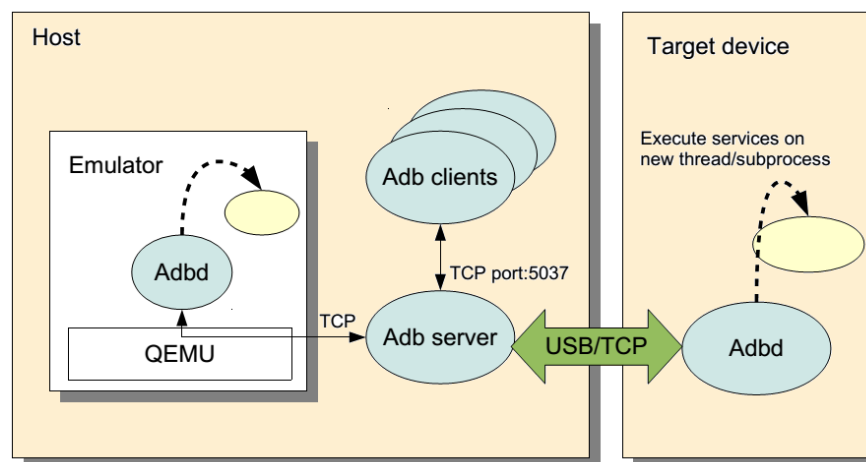
With a locked bootloader, the approach is more complicated. We should exploit possible vulnerabilities in the Android OS to gain root privilege. There are various well-known exploitations for rooting Android with a locked bootloader and most of them have been already patched. Those exploitations are documented on Common Vulnerabilities and Exposures Database (<https://cve.mitre.org/index.html>)

In this session, I analyze three case-studies that exploit the typical vulnerabilities inside the Android OS: RageAgainstTheCage, CVE-2009-1185 (Exploit), and CVE-2011-1823 (Gingerbreak).

### III.2.1 Case Study 1: Exploiting Race Condition (RageAgainstTheCage)

The idea of this exploitation is making the process of Android Debug Bridge, which has been already started with root privilege, cannot drop its privilege to normal. That kind of exploitation is well-known on the Linux-based operating systems as the name Race Condition Attack.

**Android Debug Bridge (ADB):** is a command-line tool allows a host communicate with the Android device. ADB starts as root by default and calls setuid to drop its privileges to a normal user account.



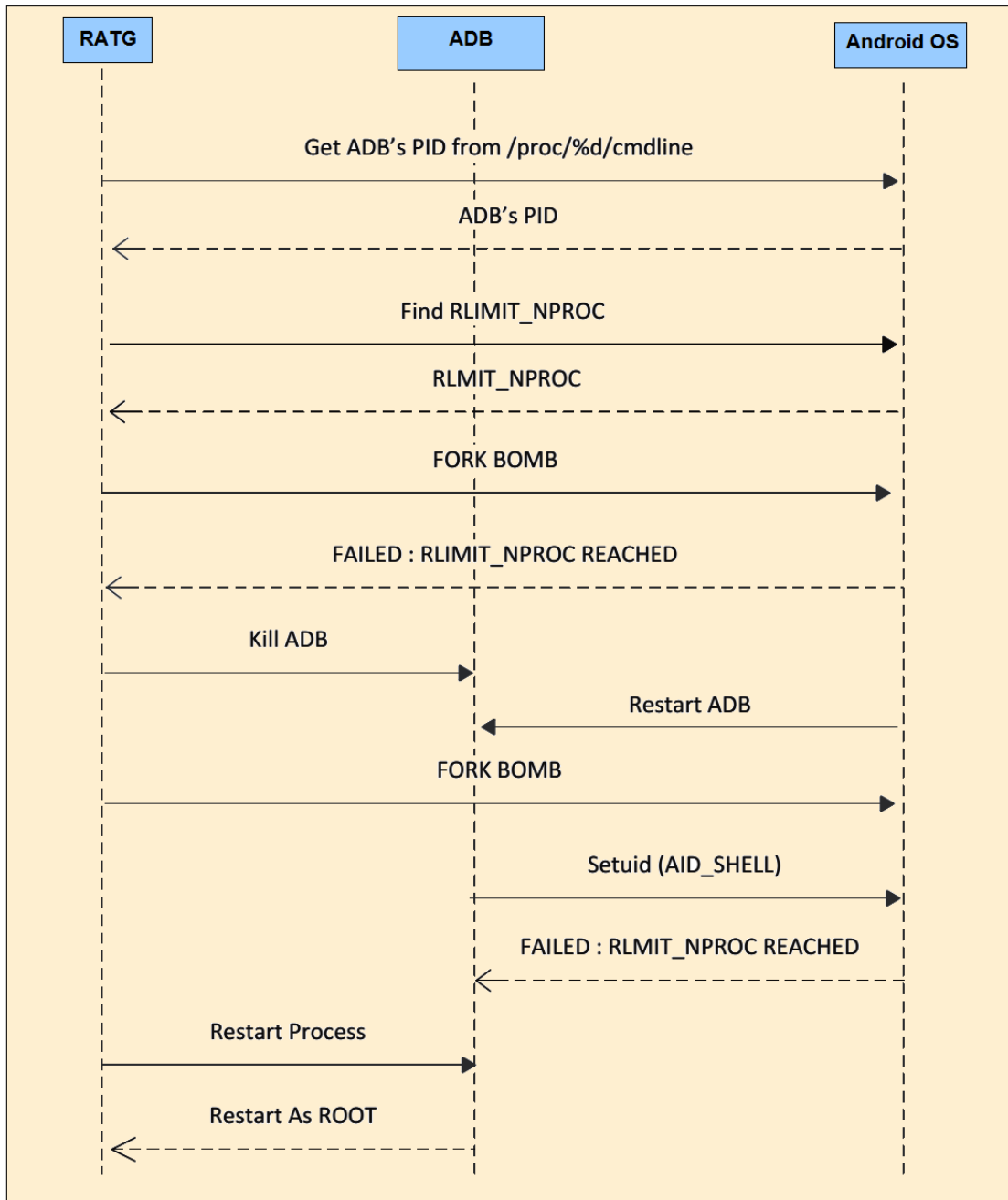
**Figure 11. The operation of Android Debug Bridge (ADB)**

**RLIMIT\_NPROC:** In Linux-based Operating System, it defines the allowed maximum number of simultaneous processes.

#### Rooting scenario:

1. RageAgainstTheCage (RATG) first checks the RLIMIT\_NPROC limit and spawns processes that do nothing until the limit is reached. This method is known as fork bomb, and it can be done by using a loop.

2. When the number of processes maxed out, RATG kills the running ADB process. Because the parent process of ADB wants to ensure that ADB is always running, it restarts the ADB process.
3. ADB must call `setuid` to drop its privileges after restarting. However, the fork bomb of RATG has already maxed out the number of running processes, therefore the call to `setuid` fails.
4. Then ADB keeps running as root. With the ADB shell running with the root privilege, we have the full control on the Android device.



**Figure 12. Rooting scenario with RageAgainstTheCage**

### III.2.2 Case Study 2: Exploiting the lack of message verification (Exploid)

The rooting method affected Android versions before 2.1 and relies on the vulnerability of udev. Briefly, udev offers dynamic management of devices. It allows standard users to “hot plug” devices that may require root level access, such as a USB device. Then udev should receive and handle various device events from the kernel. Those events are delivered to udev through NETLINK, which is a socket family (AF\_NETLINK) commonly used for IPC between userspace applications and the kernel. However, NETLINK is not exclusively for the communication between kernel and userspace. It can also be used to deliver messages between userspace applications.

There was a serious problem with the verification task in udev. As mentioned, udev (versions before 1.4.1) does not verify the source of NETLINK. Hence, the rooting job can be done by sending a NETLINK event message that triggers udev into running an arbitrary binary as root when invoking a hotplug event. To be more specific, a user space process can gain root privileges by sending an udev event claiming it originates from the kernel (which is trusted).

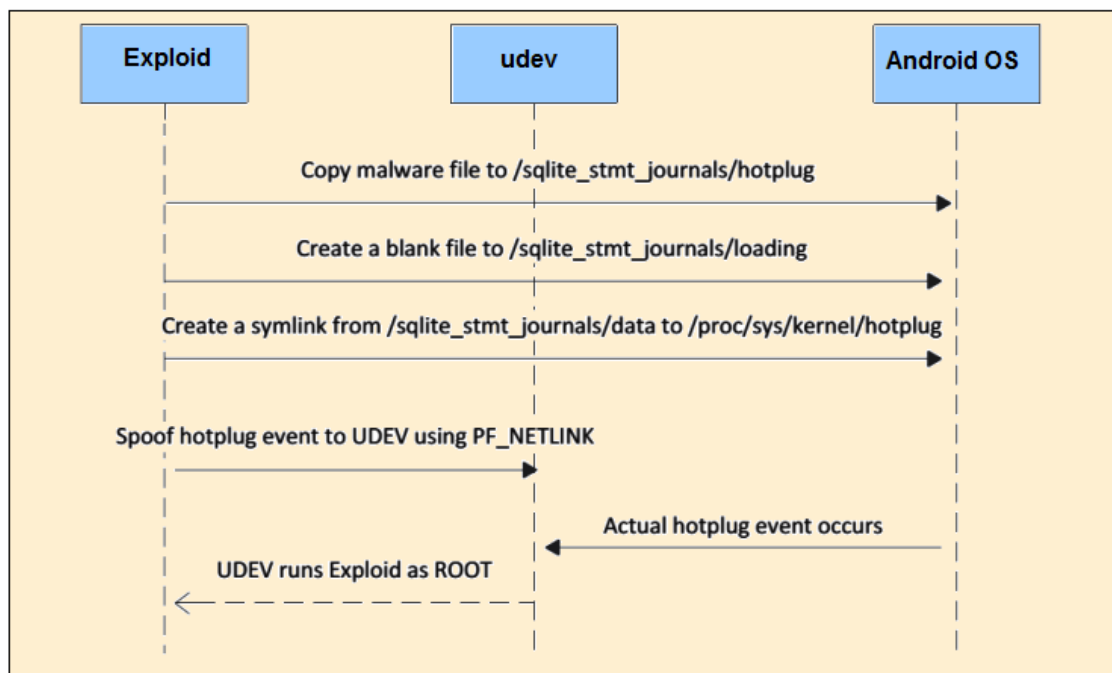


Figure 13. Rooting scenario with Exploid

#### Rooting scenario:

1. The malware **exploid** first copies itself with the name **hotplug** into a directory, for example: `/sqlite_stmt_journals/`
2. After that, it opens a socket of the domain **PF\_NETLINK** and the family **NETLINK\_KOBJECT\_UEVENT**.

According to the netlink (7) man page, **NETLINK\_KOBJECT\_UEVENT** is kernel message to user space event. Moreover, hotplug (used for managing removable media) in Linux relies on this data channel.

In mainline Linux, hotplug can detect if the user inserted an USB. In Android, the hotplug event can be triggered if the WiFi feature is turned on.



3. The malware then creates a symbolic link named `/sqlite_stmt_journals/data` from the current directory, pointing to `/proc/sys/kernel/hotplug`. After that, it sends a spoofed message to the **NETLINK** socket.

When **init** received the spoofed message and failed to validate its source, as mentioned at the beginning, it moves forward to copy the contents of the **hotplug** file to the file **data**. The task was done with the root privileges.

Hence, when the next hotplug event occurred (i.e., disconnecting and reconnecting WiFi), the kernel executed the **exploid** binary with root privilege. At that moment, the malware knew it was running with root privileges. Then, it proceeded to remount the system partition in read/write mode and created a set-uid rootshell inside the path: `/system/bin/rootshell`

### III.2.3 Case Study 3: Exploiting the signed integer check (Gingerbreak)

GingerBreak exploits the vulnerability in the Android volume manager daemon: `/system/bin/vold`.

The daemon **vold** has a method, **DirectVolume : : handlePartitionAdded**, which sets an array index using an integer passed to it. The method does a maximum length check on this integer parameter passed to it, but does not check if the integer is a positive or negative value.

Hence, if the malware sends messages that contain negative integers to **vold** via a **NETLINK** socket, the exploit code can access arbitrary memory locations. Then, the exploit code writes to the Global Offset Table (GOT) of **vold** to overwrite several functions with calls to **system()**, for example, the function **strcmp()**

Therefore, by making a call to **vold**, the malware can execute an application via **system()**, with the root privilege of **vold** (since **vold** is on the `/system` partition). If the exploit code calls **sh** and then remount `/system` as read/writable, which allows **su** to be installed.

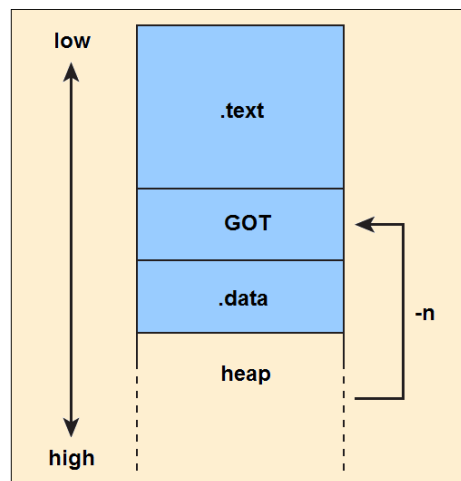


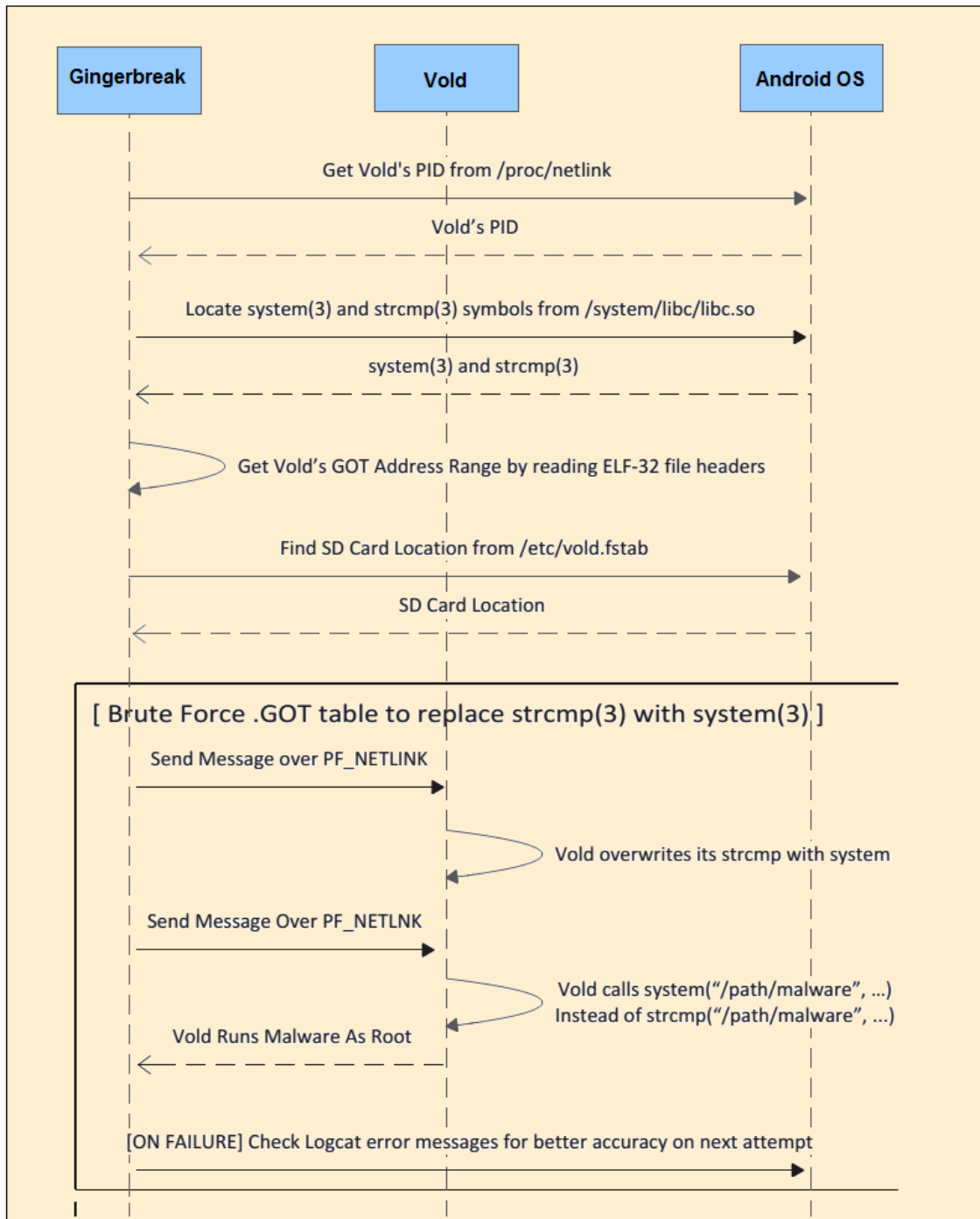
Figure 14. How a negative index can overwrite the Global Offset Table (GOT)

#### Rooting scenario:

1. First, the exploit gets the PID of the **vold** process from `/proc/net/netlink`
2. Next, it inspects the C library (`/system/libc/libc.so`) to find the symbol addresses of **system()** and **strcmp()**

3. It then parses the ELF-32 (Executable and Linkable Format) header to locate the Global Offset Table (GOT) section of vold
4. Parsing the **vold.fstab** file to find the **/sdcard** mount point in the device.

In order to get the correct negative index value, the exploit will intentionally crash the service while monitoring the result of logcat (Android log utility) output.



**Figure 15. Rooting scenario with Gingerbreak**

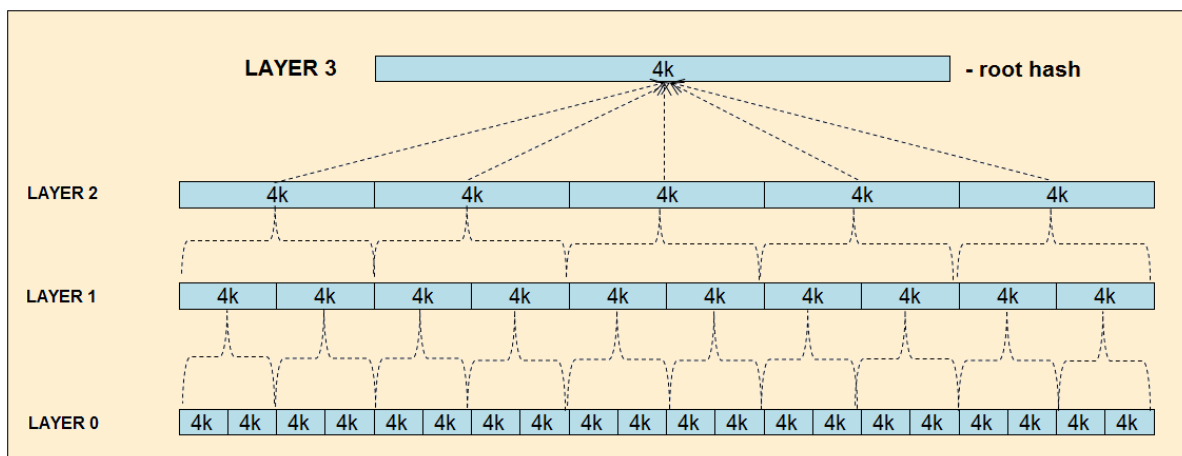
5. After having the necessary information, the exploit sends malicious **NETLINK** messages with the calculated negative index value. This causes vold to change entries in its own GOT to point to the function **system()**.
6. After one of the targeted GOT entries is overwritten, vold executes the GingerBreak binary with root privileges.

## IV. HARDENING THE OPERATING SYSTEM TO MITIGATE ROOTING

### IV.1 Ensuring boot integrity with Verified boot

Verified boot guarantees the state when booting is the same as the last used state. The feature is implemented through the optional device-mapper-verity (dm-verity) kernel feature, which provides transparent integrity checking of block devices. The dm-verity feature determines if a block device matches its expected configuration by using a cryptographic hash tree.

For every block (which size is 4k), there is a SHA256 hash. A chain of trust is established during the boot sequence through 4 layers: from the ROM, via the boot loader, onto the kernel and finally the root file system. Since the hash values are stored in a tree of pages, only the top-level root hash is trusted to verify the rest of the tree. The chance to modify any of the blocks would be equivalent to breaking the cryptographic hash.



**Figure 16. The Trust Chain of Verified Boot**

### IV.2 Light-weight Address Space Layout Randomization with Retouching

Address Space Layout Randomization (ASLR) is a defensive technique implemented at the kernel level of many mainline Linux distributions. It is an efficient and practical defense against control-flow hijacking attack (like the case of Gingerbreak). Android started providing ASLR recently (from Android 4.0 Ice Cream Sandwich) after the OS had experienced an urgent demand for better security.

The principle of ASLR is allocating memory regions (for both code and data) at random locations, in order to make the process memory layout cannot be inferred from other executions of the same program or from other co-located processes using the same shared libraries. Therefore, it can lower the chance of locating the positions of code gadgets in memory (which is essential for return-oriented programming attacks). However, pre-linking, limited processing power, and restrictive update processes make it difficult to implement ASLR on Android devices.

In the paper “**Address Space Randomization for Mobile Devices**” (Bojinov, Boneh et al. 2011), the group from Stanford and Google Android team discussed those difficulties. Since the Android mobile operating systems spend considerable effort to minimize boot time, application launch time, power consumption, and memory footprint. These optimizations make existing ASLR implementation strategies have some technical challenges:

1. The Android OS prelinks shared libraries to speed up the boot process. The prelinking happens during the build process and results in hard-coded memory addresses written in the library code. This prevents relocating these libraries in process memory. Moreover, Android uses a custom dynamic linker that cannot self-relocate at run-time (unlike ld.so). Recent attack techniques against ASLR show the need to randomize the whole process address space, including base executables and shared libraries (Roglia, Martignoni et al. 2009)
2. During normal operation, the filesystem on the Android device is mounted read-only for security reasons. This prevents binary editing tools from modifying images on the device or in file-backed memory.

The group introduced a new technique for implementing ASLR in the resource-constraint Android environment called **Retouching**, it can randomize all executable code including libraries (and prelinked libraries), base executables, and the linker but requires no kernel modifications. Retouching is conceptually similar to the rebase utility in Windows (<http://www.drdbbs.com/rebasing-win32-dlls/184416272>), which allows manually moving the starting offset of an executable or DLL file by executing relocation in advance.

The implementation concept is: compilers like GCC can generate code which is position-independent (PIC). The PIC object files have all of their location-sensitive offsets listed in relocation sections (these lists are later used to fix the library to a location at load time). There are important properties: PIC binaries can be rebased even after prelinking and binaries can be trivially reverted to their original state to support software updates. Also, randomization is possible at software update time (rather than on boot time). This enables a light-weight, user-space implementation of ASLR by preserving prelinking benefits and applying randomization during update.

### IV.3 Multi-Level Privilege Escalation Prevention with ASF and ASM

#### ASF - Extensible Multi-Layered Access Control

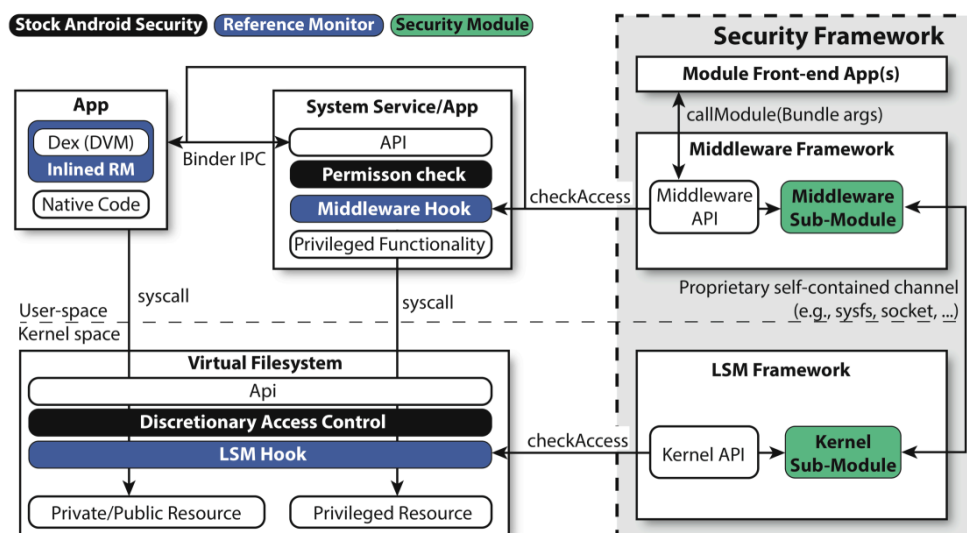


Figure 17. ASF Architecture. Adapted from (Backes, Bugiel et al. 2014)

Privilege escalation is prevented by the two most important building blocks of the ASF Architecture:

**Reference Monitors:** This multi-tiered enforcement is used at all of three layers (application layer, middleware layer, and kernel layer). Each reference monitor protects one specific privileged resource and mediates all access to the resource through the Android API.

**Security Modules:** Including kernel sub-module and middleware sub-module, each module implements a policy engine that manages its own security policies and acts as a policy decision-making point. Modules should be signed to guarantee their integrity, and the verification key is embedded in the kernel. A kernel module performs policy checks to verify that a user-space process is sufficiently privileged to issue commands to it.

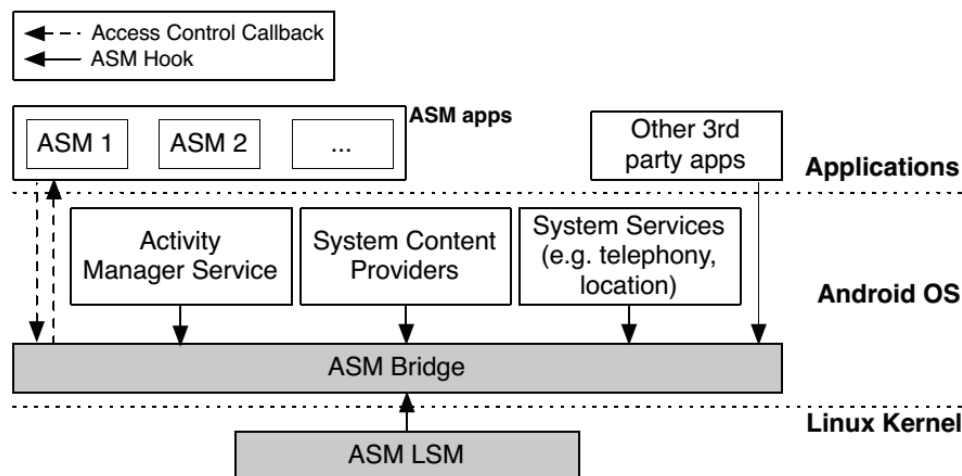
Kernel-level policies form truncation automata that terminate illegal control flows. The editing of automata policies (for example, modification of return values instead of only deny/allow decision) is supported by hooks. Middleware-level has 136 functions for enforcement hooks in all major system services and apps.

However, there is a performance trade-off for the security purpose of ASF. The overhead caused by hooks (the marshalling, sending, and unmarshalling of the hooks' parameters) constitutes 11.8% overhead.

### ASM - Android Security Modules

ASM was introduced in the paper “**ASM: A Programmable Interface for Extending Android Security**” (Heuser, Nadkarni et al. 2014). Briefly, ASM is a framework that creates a series of “authorization hooks” which define new reference modules for the Android OS. Those hooks come in five different categories: Lifecycle Hooks, OS Service Hooks, OS Content Provider Hooks, Third-Party App Hooks, LSM Hooks.

As shown in the below figure, the ASM framework implements reference monitors as ASM apps; each app has to register for a unique set of authorization hooks and callback. This security model provides a better approximation of least privilege, which limits the impact of a rooting exploitation.



**Figure 18. ASM Architecture. Adapted from (Heuser, Nadkarni et al. 2014)**

The ASM LSM layer is based on Linux Security Modules (LSM). It allows ASM apps to interact with the LSM hooks, without overriding the available security conventions. The ASM monitor interface is contained in the ASM Bridge layer, which receives and coordinates protection events from authorization hooks.

## V. CONCLUSION AND ACKNOWLEDGEMENT

In this research, I have surveyed and comparatively analyzed the Android rooting problem under the Operating System study viewpoint. With this snapshot of the overall Operating System research landscape, I hope to uncover a part of how the Android Operating System achieves its security objectives.

I would like to take this opportunity to express my heartfelt thanks to Professor Weng-Fai Wong. It has been an honor to be his student. The class CS5250 that I was very fortunate to attend in was a pleasant, valuable and memorable experience of my last semester in the National University of Singapore.

## VI. REFERENCES

1. Backes, M., et al. (2014). Android security framework: extensible multi-layered access control on Android. Proceedings of the 30th Annual Computer Security Applications Conference. New Orleans, Louisiana, USA, ACM: 46-55.
2. Backes, M., et al. (2014). XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Proceedings of the 30th Annual Computer Security Applications Conference. New Orleans, Louisiana, USA, ACM: 36-45.
3. Bojinov, H., et al. (2011). Address space randomization for mobile devices. Proceedings of the fourth ACM conference on Wireless network security. Hamburg, Germany, ACM: 127-138.
4. Drake, J. J., et al. (2014). Android Hacker's Handbook, Wiley Publishing.
5. Enck, W., et al. (2009). "Understanding Android Security." IEEE Security & Privacy 7(1): 50-57.
6. Heuser, S., et al. (2014). ASM: a programmable interface for extending android security. Proceedings of the 23rd USENIX conference on Security Symposium. San Diego, CA, USENIX Association: 1005-1019.
7. Roglia, G. F., et al. (2009). Surgically Returning to Randomized lib(c). Proceedings of the 2009 Annual Computer Security Applications Conference, IEEE Computer Society: 60-69.
8. Shao, Y., et al. (2014). "RootGuard: Protecting Rooted Android Phones." Computer 47(6): 32-40.
9. Shen, Y., et al. (2016). Insights into rooted and non-rooted Android mobile devices with behavior analytics. Proceedings of the 31st Annual ACM Symposium on Applied Computing. Pisa, Italy, ACM: 580-587.
10. Sun, S.-T., et al. (2015). Android Rooting: Methods, Detection, and Evasion. Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices. Denver, Colorado, USA, ACM: 3-14.
11. Zhang, H., et al. (2015). Android Root and its Providers: A Double-Edged Sword. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Denver, Colorado, USA, ACM: 1093-1104.
12. A look at ASLR in Android Ice Cream Sandwich 4.0: <https://duo.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0>
13. Android Verified Boot: <https://source.android.com/security/verifiedboot/>