Yet Another Android Rootkit

/protecting/system/is/not/enough/

Tsukasa OI – Research Engineer
Fourteenforty Research Institute, Inc.

Abstract

Gaining *root* privileges in Android operating system is considered as a goal for attackers but it is far from the end of the story. Attackers must understand how Android work to gain privileges inside Android application system and break security mechanisms.
On the other hand, some Japanese smartphones have additional protections (NAND lock, secure boot, integrity verification and LSM) to prevent these *root*ing issues. However, because vendors do not really understand what to protect, there is a possibility to bypass those protection mechanisms.
This paper describes how to implement an Android user-mode rootkit to take over the whole application system and what caused this "issue".

Introduction: Android and *root*ing

Privilege system is a characteristic on recent mobile platforms including Android. No applications can exercise administrative rights and all applications are separated using sandbox. It improves security dramatically but it also means customizing devices is restricted. So *root*ing, gaining *root* privileges inside Android devices came out. And for some users, it became the way to unchain their devices. However, the problem is they exploit local vulnerabilities which malware can also utilize.
DroidDream is one of the Android malware utilizing *root* exploits. It uses two exploits (called exploid and rage against the cage) to gain *root* privileges and installs its payload to the system partition which normal applications cannot write to it. As a result, this payload is impossible to remove for normal users. Even if they use factory reset, it won't work as well.

Breaking Security from *root*

There are two major security mechanisms which can be beaten by *root* privileges. The permission system and the priority associated with Activity[1].

Permission System

Permission system is well-known security mechanism in Android OS. Application developers must declare which resources their applications use (internet connection, retrieve phone number, etc.). Unless associated permissions are declared, they cannot access those resources.

[1] Activity is a user-interface plus action element in Android OS. Application developers can associate activity with action (e.g. SEND the TEXT) using Intent Filter.

However, even if applications declare permissions, not everything is really "permitted". For instance, INSTALL_PACKAGE which allows application to install another application without any confirmation is reserved for system use. It is natural that applications installed in system partition are permitted with those permissions. But *root* processes can also bypass permission mechanism because all permissions are permitted to system processes including processes running in *root* privileges. GingerMaster is the malware utilizing this behavior indirectly[2].

Activity Priorities

Activity is the one of the most important features in the Android application system. All user-interfaces are displayed using Activity. Activities can be associated with an action by Intent Filters. For example, we can define Intent Filter which handles *send*ing *text*. Intent Filter is very flexible mechanism and we can even define the URL handler.
There's one more, Intent Filter can have priority value which determines which Activity should be handled when multiple Activities matches. It enables Activity "hooking". If an attacker defines Intent Filter with higher priority, it can hide Activities with lower priority values.
This is a dangerous behavior and priority associated with Activity cannot be higher than default value unless it is the system application. But if the attacker can install a package to the system partition, this restriction can be bypassed.

Root isn't the goal

Of course, attacker can use traditional ptrace or some legacy system calls to bypass all security. Still, this is not the end of the story because some Android devices (especially Japanese smartphones) have additional security mechanism to prevent those *root*ing issues.

Vendor-Specific Protection

Some Android device vendors have added some security mechanisms to prevent system software to be taken over or overwritten. The main reason for this is to protect their proprietary material. They do not want proprietary material to be altered or sometimes viewed. As a result, *root* user in some Android devices has very restricted rights.

NAND Lock

This is mostly implemented as a kernel-mode driver feature. It blocks all write requests to important partitions including system partition. This is not easy to bypass but an attacker can mount system directory to other mount point.

Secure Boot

This is implemented in the board or as a boot loader. It prevents unsigned (modified) boot loader or kernel to be executed. This is also not easy to bypass but it is only a boot protection. In other words

[2] GingerMaster does not install package directly. But it calls built-in "pm" program in the *root* shell. This "Package Management" command is written in Java. As a result, INSTALL_PACKAGE permission is used.

this protection does not cover on-memory modifications.

Integrity Verification

Some Japanese smartphones have integrity verification feature to prevent important processes to be tainted. It verifies the program if it is legitimate and disables some features if it is not. However, there are many holes which the attacker can bypass.

Linux Security Modules (LSM)

Linux Security Modules is a security framework implemented in the Linux kernel. It is famous that this is used by SELinux and more of that, some Japanese smartphones use LSM to protect Android system. The most famous example is Miyabi LSM developed by Sharp Corp.
Miyabi LSM protects important mount points (e.g. /system). It prevents modification of the system. This LSM also disables some system calls (e.g. *ptrace*) to prevent system to be taken over.
LSM is a strong mechanism and if LSM is implemented properly, it is nearly impossible to take over the system. But making "proper" protection policy is very difficult. Developers have to understand **everything** running inside Android operating system while implementation. Unfortunately, LSM developers didn't. This could lead to LSM bypassing.

Conclusions

Additional protections are strong mechanisms. However, it is not impossible to bypass everything. Even if the attacker cannot write to system partition (/system), he/she can make a user-mode rootkit and take over the system.

Android User-Mode Rootkit

Before implementing my Android rootkit, I made some restrictions.

- No kernel mode
- No /dev/*mem or /proc/*/mem
- No *ptrace*, *chroot*, *pivot_root*...
- No writes to system partition (/system)

Even with this restriction, an Android user-mode rootkit can still be implemented by bypassing all current vendor-specific protections. The attacker only needs *root* privilege.

Tainting Zygote

Android OS has important process called Zygote. This process hosts Dalvik VM and all normal Android applications are cloned (*fork*ed) from this process. This is the point that Zygote daemon (running in *root* privileges) accepts *fork* requests through UNIX-domain socket.
First, the rootkit needs to kill Zygote to force Zygote daemon to be restarted. Zygote initialization has a race-condition which enables attackers to replace UNIX-domain socket. The rootkit exploits this behavior. By replacing UNIX-domain socket, the rootkit performs man-in-the-middle attack. Now the rootkit can modify all fork requests so the rootkit then modifies requests to load the rootkit's

payload (written in Java).

Modifying Dalvik VM state

Dalvik VM manages its state on-memory. The global state is available on the gDvm symbol in libdvm.so. Modifying this enables rootkit hook injection.
To achieve this, the rootkit payload must have ability to read/write arbitrary memory. So the rootkit's payload uses sun.misc.Unsafe class for this purpose and enables memory read/write without any native code.
One of the easiest ways to inject hook is class replacement. gDvm holds all loaded class information inside a member structure called loadedClasses. Modifying this, the attacker can swap two existing (loaded) classes. It means replacing existing class with a malicious one.
I implemented a simple rootkit to replace WebView class (which is web browser class) to monitor browser's activities. It could be used to hook everything inside all Android applications.

Bottom Line

Another Issue – Overprotection

There is one more issue and that is overprotection. Some protection mechanisms will make incompatibility. One good example is Miyabi LSM. It rejects *ptrace* system call but it has serious side effects. Application developers cannot even debug **their** native applications. I think this is overprotection and also an issue because overprotection may make Android system closed.

Conclusion – So what was wrong?

This rootkit is not really advanced and not even completely stealthy. Developers could easily make rules to prevent this type of attack but this is not the point. Currently, Google did not published guidelines to protect Android system. I think this is a problem. Vendors have to make implementation on their own which may cause overprotection and/or insufficient protection. I suggest sharing information (including protection guidelines) to prevent Android system to be taken over.