Chuan Ji

October 19, 2011 · ANDROID TECHNICAL NOTES

How Rooting Works: A Technical Explanation of the Android Rooting Process

I have always been curious how rooting actually worked behind the scenes. After recently acquiring a new Eee Pad Slider, a Honeycomb tablet that so far no one has been able to root, the frustration of being locked out of this amazing piece of hardware with so much potential led me to finally sit down and figure out what exactly rooting means, what it entails from a technical perspective, and how hackers out in the wild are approaching the rooting of a new device. Although all this information is out there, I have not been able to find a good article that had both the level of technical detail that I wanted and an appropriate introduction to the big picture, and so I decided to write my own.

This is NOT a noob-friendly guide to rooting a particular Android device. Rather, it is a general explanation of how stock Android ROMs try to prevent unprivileged access, how hackers attack this problem and how

rooting software leverage various exploits to defeat these security mechanisms.

I. The Goal

Let us first take a step back and consider *exactly* what we mean by rooting. Forget flashing custom ROMs, enabling WiFi tethering or installing Superuser.apk; fundamentally, rooting is about obtaining root access to the underlying Linux system beneath Android and thus gaining absolute control over the software that is running on the device. Things that require root access on a typical Linux system — mounting and unmounting file systems, starting your favorite SSH or HTTP or DHCP or DNS or proxy servers, killing system processes, chroot-ing, etc., — require root access on Android as well. Being able to run arbitrary commands as the root user allows you to do absolutely *anything* on a Linux / Android system, and this is real goal of rooting.

Stock OEM Android builds typically do not allow users to execute arbitrary code as root. This essentially means that you as a user are granted only limited control over your own device; you can make your device do task *X* only if the manufacturer explicitly decided to allow it and shipped a program to do it. You will not be able to use third-party apps to accomplish a task that your manufacturer does not wish you to do. WiFi tethering is a good example of this. Cell phone carriers obviously do not want you to tether your

phone without paying them additional charges. Therefore, many phones come pre-packaged with their own proprietary WiFi tethering apps that demand extraneous fees. But without root access, you will not be able to install a free alternative like Wireless Tether For Root Users. Why this is accepted practice in the industry is a mystery to me. The only difference between cell phones, tablets and computers is their form factor; but while a PC vendor would fail spectacularly if they tried to prevent users from running arbitrary programs on their machines, cell phone vendors are clearly not judged along the same lines. But such arguments would belong to another article.

II. The Enemy: Protection Mechanisms On A Stock OEM Android ROM

Bootloader and Recovery

The bootloader, the first piece of code executed when your device is powered on, is responsible for loading the Android OS and the recovery system and flashing a new ROM. People refer to some bootloaders as "unlocked" if a user can flash and boot arbitrary ROMs without hacking; unfortunately, many Android devices have locked bootloaders that you would have to hack around in order to make them do anything other than boot the stock ROM. A Samsung smartphone I had used some

months ago had an unlocked bootloader; I could press a certain combination of hardware keys on the phone, connect it to my computer, and flash any custom ROM onto it using Samsung's utilities without having to circumvent any protection mechanisms. The same is not true for my Motorola Droid 2 Global; the bootloader, as far as I know, cannot be hacked. The Eee Pad Slider, on the other hand, is an interesting beast; as with other nVidia Tegra 2 based devices, its bootloader is controllable through the `nvflash` utility, but only if you know the *secure boot key* (SBK) of the device. (The SBK is a private AES key used to encrypt the commands sent to the bootloader; the bootloader will only accept the command if it has been encrypted by the particular key of the device.) Currently, as the SBK of the Eee Pad Slider is not publicly known, the bootloader remains inaccessible.

System recovery is the second piece of low-level code on board any Android device. It is separate from the Android userland and is typically located on its own partition; it is usually booted by the bootloader when you press a certain combination of hardware keys. It is important to understand that it is a totally independent program; Linux and the Android userland is not loaded when you boot into recovery, and any high-level concept such as root does not exist here. It is simple program that really is a very primitive OS, and it has absolute control over the system and will do anything you want as

Chuan Ji

long as the code to do it is built in. Stock recovery varies with the manufacturer, but often includes functionalities like reformatting the `/data` partition (factory reset) and flashing an update ROM (`update.zip`, located at the root of the external microSD card) signed by the manufacturer. Note I said *signed by the manufacturer*; typically it is not possible to flash custom update files unless you obtain the private key of the manufacturer and sign your custom update with it, which is both impossible for most and illegal under certain jurisdictions. However, since recovery is stored in a partition just like `/system`, `/data` and `/cache` (more about that later), you can replace it with a custom recovery if you have root access in Linux / Android. Most people do just that upon rooting their device; ClockworkMod Recovery is a popular third-party recovery image, and allows you to flash arbitrary ROMs, backup and restore partitions, and lots of other magic.

ADB

ADB (see the official documentation for ADB) allows a PC or a Mac to connect to an Android device and perform certain operations. One such operation is to launch a simple shell on the device, using the command `adb shell`. The real question is what user do the commands executed by that shell process run as. It turns out that it depends on the value of an Android system property, named `ro.secure`. (You can view

the value of this property by typing `getprop ro.secure` either through an ADB shell or on a terminal emulator on the device.) If `ro.secure=0`, an ADB shell will run commands as the root user on the device. But if `ro.secure=1`, an ADB shell will run commands as an unprivileged user on the device. Guess what `ro.secure` is set to on almost every stock OEM Android build. But can we change the value of `ro.secure` on a system? The answer is no, as implied by the `ro` in the name of the property. The value of this property is set at boot time from the `default.prop` file in the root directory. The contents of the root directory are essentially *copied* from a partition in the internal storage on boot, but you cannot write to the partition if you are not already root. In other words, this property denies root access via ADB, and the only way you could change it is by gaining root access in the first place. Thus, it is secure.

Android UI

On an Android system, all Android applications that you can see or interact with directly are running as _un_privileged users in sandboxes. Logically, a program running as an unprivileged user cannot start another program that is run as the privileged user; otherwise any program can simply start another copy of itself in privileged mode and gain privileged access to everything. On the other hand, a program running as root can start another program as root or as an

unprivileged user. On Linux, privilege escalation is usually accomplished via the `su` and `sudo` programs; they are often the only programs in the system that are able to execute the system call `setuid(0)` that changes the current program from running as an unprivileged user to running as root. Apps that label themselves as requiring root are in reality just executing other programs (often just native binaries packaged with the app) through `su`. Unsurprisingly, stock OEM ROMs never come with these `su`. You cannot just download it or copy it over either; it needs to have its SUID bit set, which indicates to the system that the programs this allowed to escalate its runtime privileges to root. But of course, if you are not root, you cannot set the SUID bit on a program. To summarize, what this means is that any program that you can interact with on Android (and hence running in unprivileged mode) is unable to either 1) gain privileged access and execute in privileged mode, or 2) start another program that executes in privileged mode. If this holds, the Android system by itself is pretty much immune to privilege escalation attempts. We will see the loophole exploited by on-device rooting applications in the next section.

III. Fighting the System

So how the hell do you root an Android? Well, from the security mechanisms described

above, we can figure out how to attack each component in turn.

If your device happens to have an unlocked bootloader, you're pretty much done. An example is the Samsung phone that I had had. Since the bootloader allowed the flashing of arbitrary ROMs, somebody essentially pulled the stock ROM from the phone (using `dd`), added `su`, and repackaged it into a modified ROM. All I as a user needed to do was to power off the phone, press a certain combination of hardware keys to start the phone in flashing mode, and use Samsung's utilities to flash the modified ROM onto the phone.

Believe it or not, certain manufacturers don't actually set `ro.secure` to 1. If that is the case, rooting is even easier; just plug the phone into your computer and run ADB, and you now have a shell that can execute any program as root. You can then mount `/system` as read-write, install `su` and all your dreams have come true.

But many other Android devices have locked bootloaders and `ro.secure` set. As explained above, they should not be root-able because you can only interact with unprivileged programs on the system and they cannot help you execute any privileged code. So what's the solution?

We know that a number of important programs, including low-level system services, must run as root even on Android in order to access hardware resources. Typing

`ps` on an Android shell (either via ADB or a terminal emulator on the device) will give you an idea. These programs are started by the `init` process, the first process started by the kernel (I often feel that the kernel and the `init` process are kind of analogous to Adam and Eve — the kernel spawns `init` in a particular fashion, and `init` then goes on and spawns all other processes) which has to run as root because it needs to start other privileged system processes.

Now here's the key insight: if you can hack / trick one of these system processes running in privileged mode to execute your arbitrary code, you have just gained privileged access to the system. This how all one-click-root methods work, including z4root, gingerbreak, and so on. If you are truly curious, I highly recommend this excellent presentation on the various exploits used by current rooting tools, but the details are not as relevant here as the simple idea behind them. That idea is that there are vulnerabilities in the system processes running as root in the background that, if exploited, will allow us to execute arbitrary code as root. Well, that "arbitrary code" is most certainly a piece of code that mounts `/system` in read-write mode and installs a copy of `su` permanently on the system, so that from then on we don't need to jump through the hoops to run the programs we really wanted to run in the first place.

Since Android is open source as is Linux, what people have done is to scrutinize and

reason about the source code of the various system services until they find a security hole they can leverage. This becomes increasingly hard as Google and the maintainers of the various pieces of code fix those particular vulnerabilities when they are discovered and published, which means that the exploits will eventually become obsolete with newer devices. But the good news is that manufacturers are not stupid enough to push OTA updates to fix a vulnerability just to prevent rooting as it is very expensive for them; in addition, devices in the market are always lagging behind the newest software releases. Thus, it takes quite some time before these rooting tools are rendered useless by new patches, and by then hopefully other exploits would have been discovered.

IV. See It In Action!

To see all of this in action, you are invited to check out my follow-up article: Android Rooting: A Developer's Guide, which explains how I applied this stuff to figure out how to root an actual device.

ALSO ON JICHU4N.COM

10 years ago · 13 comments

How X Window Managers Work, And ...

10 years ago · 6 co

DEBUG trap PROMPT

48 Comments

Login

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

11 Share

Best Newest Oldest

/