Android Modding for the Security Practitioner

Dan Rosenberg

Who am I?

- Security consultant and vulnerability researcher at Virtual Security Research in Boston
  - App/net pentesting, code review, etc.
  - Published some bugs
  - Linux kernel exploitation
  - Rooted a few Android phones

Goals of this Talk

- Clarify terminology
- Demystify Android rooting and modding techniques
- Draw some conclusions about security impact of modding

Agenda

- The modding community
- Locked and unlocked bootloaders
- Flashing
- Case studies in rooting
- Post-root hacks

The Modding Community

Why Do People Want to Mod?

- Expert usage
  - Root-privileged applications for backup
  - Tethering
  - Overclocking/underclocking

- Customization
  - Custom ROMs, themes
  - Removal of bloatware

Why Do People Want to Mod?

- Upgradeability
  - Cheap, subsidized phones -> phones become obsolete rapidly -> carriers halt support
  - Modding allows continued upgrades (security and otherwise) in the event of missing carrier support

- Freedom
  - Full control over your own hardware

VSR

The Modding Community

- Modding community is largely Android enthusiasts with varying levels of technical background
  - Result: mixed or confusing terminology, lack of consistent definitions of terms

- Dozens of Android forums and publications
  - Most popular: XDA Developers, RootzWiki, AndroidForums

Why Don't (Some) Carriers Want You Modding?

- Support costs (tech support, warranty claims for bricked devices)
- Removal of sources of advertising revenue
- Free tethering conflicts with business model
- Ambiguous claims about “security”
  - We'll take a look at this one

What Prevents People from Modding?

- Two primary prevention strategies:

- OS protections
  - Prevent users from gaining root (administrative) access on their devices

- Hardware/firmware protections
  - Prevent users from flashing new firmware images

Locked and Unlocked Bootloaders

What is a “Locked” Bootloader

- Term has come to encompass a variety of restrictions preventing customization
- My definition: “A bootloader that performs cryptographic signature verification to prevent booting custom, non-signed code”
- Implementation will vary based on vendor

The State of Unlocked Bootloaders

- Wide variety of tablet OEMs (Toshiba, ASUS, Lenovo, Sony)
- Four biggest phone OEMs: Samsung, Motorola, HTC, LG
- Varied degrees of bootloader locking
  - Samsung ships mostly unlockable
  - HTC supports official unlocking (voids warranty)
  - LG ships unlocked, but no default flashing support
  - Motorola tends to be locked tight, no custom ROMs and no downgrading

How Do Locked Bootloaders Work?

- Varies by hardware implementation

- Basic idea:
  - On-chip crytographic verification of early stage bootloader
  - Bootloader verifies signature of subsequent stage before loading (kernel, Android recovery, etc.)

- If signature check fails, drops into a failsafe mode for recovery

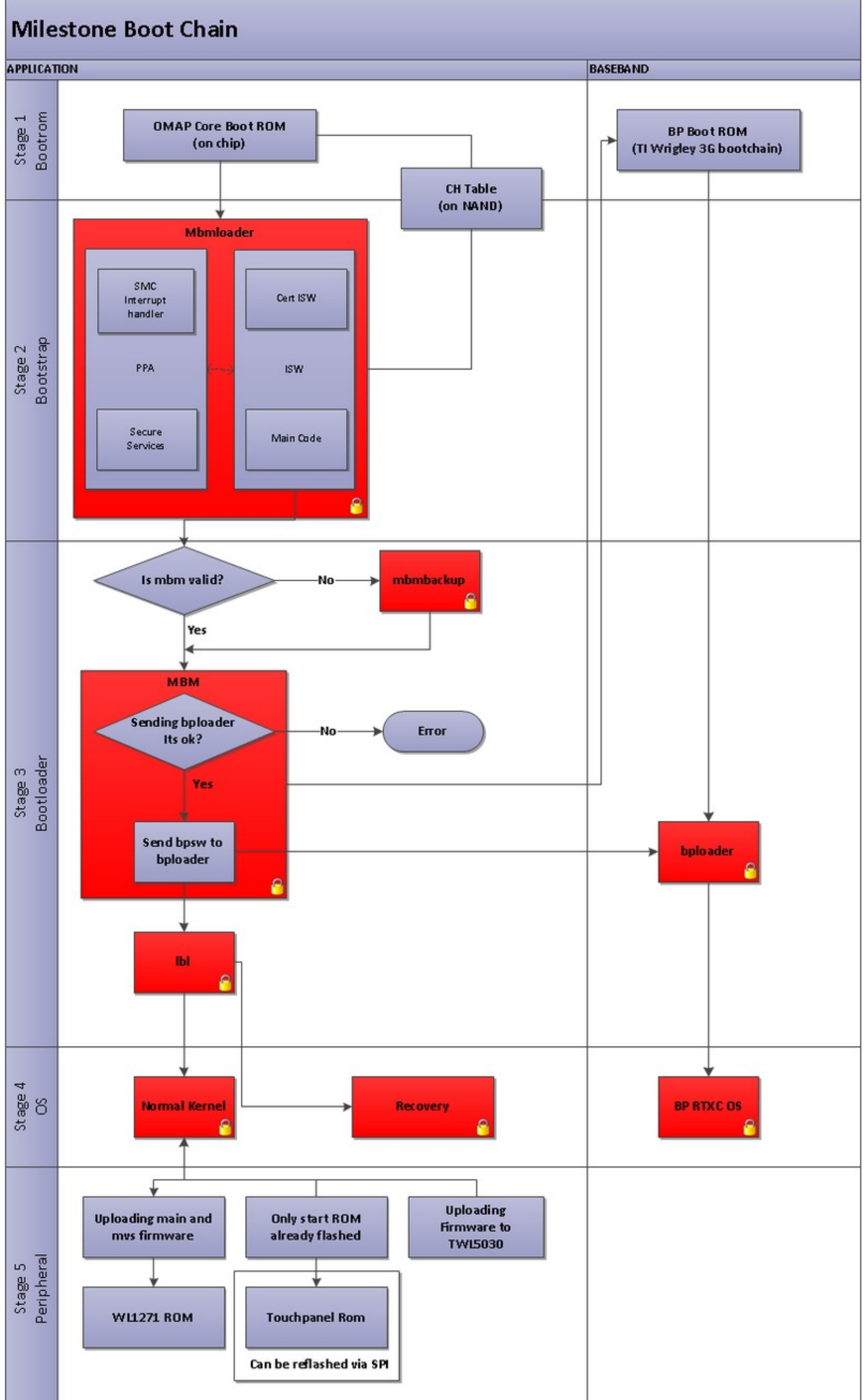
Android Partition Layout

- Actual partitions will vary by manufacturer and chipset
- Relevant to Android operating system:
  - ***system***: binary applications, system configuration, services
  - ***userdata***: user-installed apps, contacts, data
  - ***boot***: kernel, filesystem root
  - ***recovery***: Android recovery system
  - ***cache***: various frequently accessed system data
  - ***misc***: odds and ends

Case Study: Motorola/OMAP

- SHA1 hash of root public key stored in eFUSE
- Boot ROM verifies hash of key stored in mbmloader and signature on mbmloader
- mbmloader verifies signature on mbm (“Motorola Bootloader Mode?”)
- mbm verifies signature on lbl (“Linux Boot Loader”)
- lbl verifies signature on normal kernel or recovery

VSR


Milestone Boot Chain
APPLICATION
BASEBAND
Stage 1 Bootrom
Stage 2 Bootstrap
Stage 3 Bootloader
Stage 4 OS
Stage 5 Peripheral
OMAP Core Boot ROM (on chip)
CH Table (on NAND)
BP Boot ROM (TI Wrigley 3G bootchain)
Mbmloader
SMC Interrupt handler
PPA
Secure Services
Cert ISW
ISW
Main Code
Is mbm valid?
No
mbmbackup
Yes
MBM
Sending bploader Its ok?
No
Error
Yes
Send bpsw to bploader
bploader
lbl
Normal Kernel
Recovery
BP RTXC OS
Uploading main and mvs firmware
Only start ROM already flashed
Uploading Firmware to TWL5030
WL1271 ROM
Touchpanel Rom
Can be reflashed via SPI

Case Study: HTC/Qualcomm

- Primary processor (baseband) executes Primary Boot Loader (PBL) from ROM

- If FORCE_TRUSTED_BOOT Qfuse blown, verify signature of Secondary Boot Loader (SBL)
  - Public key stored via Qfuse

- SBL verifies signature on REX/AMSS (baseband) and HBOOT (app processor bootloader), starts app processor running HBOOT

- HBOOT verifies signature on kernel/recovery, boots into operating system

HTC S-ON/S-OFF

- On some HTC devices, NAND lock prevents writing to system, kernel, and recovery partitions (“S-ON”)
- Flag in radio NVRAM (“@secuflag”) is checked by HBOOT, which enforces NAND lock
- Unsetting @secuflag or providing HBOOT that does not enforce is required to flash custom ROMs (“S-OFF”)
- Created distinction between temporary root (“temp root”) and permanent root (“perm root”, “perma-root”)
  - You'll hear these terms misused outside of HTC, where they are meaningless

HTC Bootloader Unlocking

- Submit device-specific token to HTC
  - Voids warranty
- Download and flash signed binary blob
- HBOOT verifies blob and sets flag
  - Disables signature checking on kernel, recovery, and system

Fastboot Bootloader Unlocking

- If device is unlockable, just say the magic words:
  - “fastboot oem unlock”
  - We'll talk about fastboot in a bit

- Disables signature checks on all partitions

- Wipes userdata partition
  - Important for data protection
  - Otherwise, could flash compromised kernel/system/recovery and steal user data

Flashing

More Fragmentation

- Many proprietary and open flashing protocols
- Vary by both handset manufacturer and chipset
- Terms are used interchangeably by Android modding community, leading to confusion

Fastboot

- Standardized Android protocol for flashing over USB
  - “Client” is fastboot utility from AOSP
  - “Server” is proprietary OEM-specific implementation in second-stage bootloader

- Flashes full disk images to specific partitions
  - Any signature checking happens at boot, not at flashing

- Many phones disable for security reasons

Update.zip

- Officially supported Android update mechanism
- Implemented in Android recovery
- Copy zip file to SD card or internal storage
  - Full binaries, or binary diff
- Validates RSA signature against manufacturer keys
- Bugs in the past
  - Original Droid root

APX Mode / nvflash

- Tegra devices only

- Implemented in boot ROM

- All communication is AES-128-CBC encrypted
  - Uses Secure Boot Key (SBK)
  - Implemented in hardware as blown fuses
  - Some SBKs are public or based on device ID
  - Others are OEM secrets

- Upload "miniloader", a minimal bootloader, that handles actual flashing

SBF

- Motorola proprietary format
- Similar to nvflash, but implemented in secondary bootloader (“mbm”) instead of in boot ROM
- Client uses RSD Lite (“Remote Software Download”)
- Upload minimal bootloader to handle actual flashing
  - Miniloader is signature-checked
- Since Droid 3, replaced by Fastboot

Misc. Custom Tools/Protocols

- KDZ
  - LG download mode
- Odin
  - Samsung download mode
- PDL
  - Pantech download mode
- RUU (ROM Upgrade Utility)
  - HTC utility, just a Fastboot wrapper

Flashing and Data Protection

- Userdata partition contains everything valuable
  - Contacts, mail, SMS, apps, app data

- All flashing protocols reachable prior to booting OS
  - Device passcode won't save you

Flashing and Data Protection

- Without disk encryption, all data is recoverable if:
  - SBK of a Tegra device is leaked or predictable
    - Use nvflash to read userdata
  - Bootloader is kept unlocked
    - Flash compromised recovery/kernel/system, boot, read from userdata block device
- With disk encryption, bootloader status has no effect on data protection
  - ...if you actually require a strong password

Rooting

Why Root?

- Need root access to operating system to perform administrative tasks
- It's possible to have a device that:
  - Has unlocked bootloader (can **boot** unsigned code)
  - Does not allow **flashing** unsigned code
- In these cases, custom ROMs are only possible after gaining root and writing to block devices directly
- On devices with locked bootloaders, need root to customize anything

Background: Android Debugging Bridge (ADB)

- Connect over Wifi or USB
  - Enabled in device settings ("USB Debugging Mode")

- Allows installing applications

- ADB shell has uid/gid "shell", and lots of groups:

```
/* add extra groups:
** AID_ADB to access the USB driver
** AID_LOG to read system logs (adb logcat)
** AID_INPUT to diagnose input issues (getevent)
** AID_INET to diagnose network issues (netcfg, ping)
** AID_GRAPHICS to access the frame buffer
** AID_NET_BT and AID_NET_BT_ADMIN to diagnose bluetooth (hcidump)
** AID_SDCARD_RW to allow writing to the SD card
** AID_MOUNT to allow unmounting the SD card before rebooting
*/
```

Background: Android Properties

- Android uses “property” system for system settings
- Applications can set arbitrary properties, except reserved property namespaces
- “ro” (read-only) properties can only be set once, never changed

ADB + Properties = ?

- Certain properties have special meaning to ADB
- If "`ro.secure`" is 0, ADB shell runs as root
- Lesser known: if "`ro.kernel.qemu`" is 1, ADB shell runs as root:

```
/* run adbd in secure mode if ro.secure is set and
 ** we are not in the emulator
 */
```

Case Study: Motofail

VSR

The Goal

- The Android init process parses `/data/local.prop` for property settings at boot
- If we can modify this file to set any of those “special” properties, we win, because ADB shell will run as root
- Fortunately, there are lots of file permission bugs :-)

Motofail: The Bugs

- Motorola init.rc script (run as root) had multiple bugs:

```
mkdir /data/dontpanic
chown root log /data/dontpanic
chmod 0770 /data/dontpanic
# create logger folder
mkdir /data/logger 0770 radio log
chown radio log /data/logger
chmod 0770 /data/logger
# workaround: in solana somebody deletes the logfile.
# we have to back it up.
copy /data/dontpanic/apanic_console /data/logger/last_apanic_console
```

- ADB shell has group “log”

Exploit Flow

- Put a file containing the string "`ro.kernel.qemu=1`" at `/data/dontpanic/apanic_console`
- Place a symlink pointing to `/data/local.prop` at `/data/logger/last_apanic_console`
- On reboot, `init` will copy our file on top of `local.prop`, and ADB will run as root!

Motofail: The Emulator

- Adversarial relationship between rooters and OEMs
  - Goal is to keep bugs unpatched as long as possible

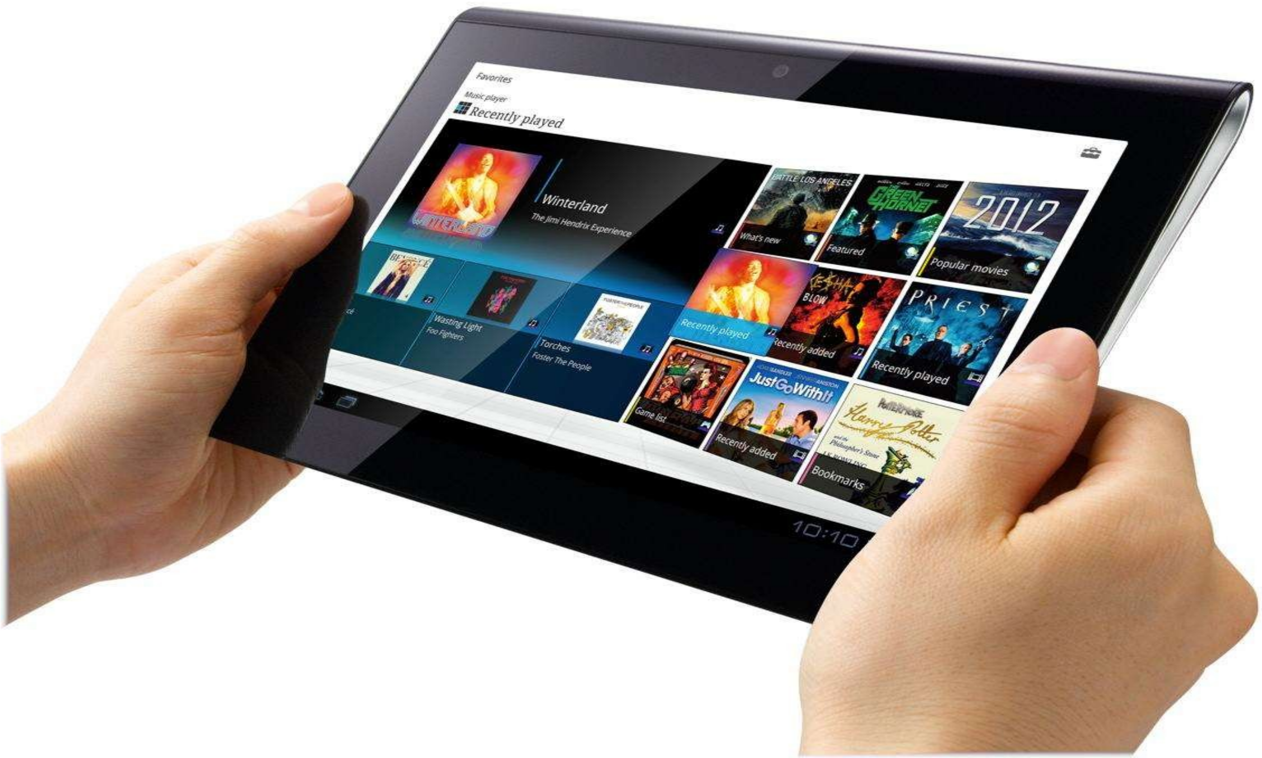
- To prevent patching, Motofail was heavily obfuscated
  - Exploit ran inside custom emulator
  - Dirty tricks to prevent dynamic analysis
  - Dummy code generation for false trails
  - Included full list of filesystem contents in binary

- Motorola fixed it quickly anyway :-(
  - Please email me if you were the one who had to reverse engineer this

Lessons from Motofail

- File permission bugs are a serious problem on Android
- Exploit is not possible without group “log”
  - This group is granted to applications that request `android.permission.READ_LOGS`
  - This permission substantially increases the attack surface exposed to malicious applications
- Disable USB Debugging mode when not in use
  - Cripples data protection if lost device is rootable

Case Study: Sony Tablet S



Sony Tablet S: The Bug

- Again, started with the obvious: `/log` directory is writable by group log

- Directory contains root-owned log files that represent on-disk copies of the Android debugging logs (logcat)

- Log backups are created with predictable filenames

- Observed that replacing log backup with a symlink and triggering a log dump by writing to logcat will:
  - Create a new file anywhere with the log contents
  - Append log contents to any existing file

Plan of Attack

- Ultimate goal: get the string "`ro.kernel.qemu=1`" into `/data/local.prop`

- On any other device, this would be easy:
  - We can partially control the log file contents by writing to logcat
  - If `local.prop` doesn't exist, vuln will create it
  - If `local.prop` does exist, vuln will append to it

- But...

OEM Customization

- On this particular device, `/data/local.prop` is a symbolic link to `/configs/local.prop`, which is a read-only filesystem (can't append)
- Need to find a way to remove existing symlink in order to create new `local.prop` file

How to Remove Arbitrary Files

- Noticed odd behavior in Android Package Manager (pm)

- pm distinguishes between "system" and "user-installed" packages
  - System apps are OEM-installed in `/system/app`

- Every app has a data directory in `/data/data/[app]/`
  - Includes `lib/` directory for native libraries
  - System apps are expected to have empty "`lib`" dirs

How to Remove Arbitrary Files, cont.

- If a system app's lib directory is not empty on boot, the Package Manager will empty it
- What happens if we replace a system app's lib directory with a symbolic link to a directory we want empty?
- pm will follow symlinks and non-recursively empty this directory!

How to Execute Code as a System App

- "`run-as`" program allows ADB shell to assume privileges of any application marked as "debuggable"
- Parses `/data/system/packages.list` file to determine status and uid of packages
- Normally, no system apps are marked debuggable
- But, we can append data to arbitrary files!
  - Modify `/data/system/packages.list` to make a system app debuggable

Putting it All Together

- Trigger log vulnerability to append fake package information to `/data/system/packages.list`
- Use "`run-as`" to assume privileges of system app
- Replace system app's lib directory with symlink to `/data`
- Reboot, `/data/local.prop` will be removed
- Use log vulnerability again to create new `local.prop`
- Reboot and run ADB as root

Lessons from Sony Root

- Root vulnerability ?= security vulnerability
  - This cannot be exploited by malicious applications
- “Benign” roots are often patched faster than real security bugs
  - Hmm...
- Multiple bugs may be chained together to achieve goal

Post-Root Modding

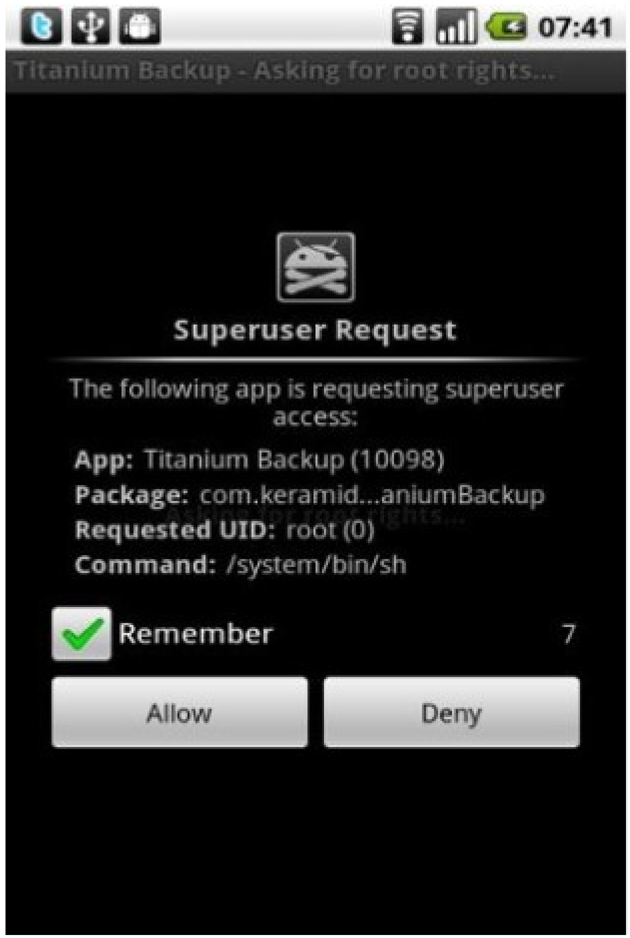
Custom Recovery Partitions

- Replaces stock Android recovery system
  - Allows easily and safely flashing custom partitions
- Most popular: ClockworkMod Recovery (CWM)
- If bootloader is locked, can't flash custom recovery
  - Instead, can hijack original recovery executable ("bootstrap recovery")

2nd Init, 2nd System, and kexec

- Unable to flash custom kernels on locked bootloaders
- 2nd Init: use ptrace() to hijack init process early and run custom init scripts
  - Allows customization of early boot process
- 2nd System: mount a custom system partition on top of original, preserving the original while allowing OS mods
- kexec: use the kexec() system call to boot into a new kernel without flashing to disk

How is Root Access Provisioned?



VSR

Su and Superuser

- No passwords to type in
- “su” is setuid root native binary
- “Superuser” is Android APK (application)
- Applications execute su to gain root privileges
- su communicates with Superuser over Unix socket to check database of permitted apps/uids
  - Permit, deny, or prompt based on response

VSR

How Su Increases Attack Surface

- By default, no setuid binaries accessible by apps
- Just the presence of setuid binaries can enable exploitation of privilege escalation vulnerabilities
- CVE-2010-3847, CVE-2010-3856
  - Tavis Ormandy's glibc vulns, require setuid to exploit
- CVE-2012-0056
  - “Mempodroid” exploit, requires setuid app

Evaluating su

- User "shell" and "root" automatically permitted:

```
if (su_from.uid == AID_ROOT || su_from.uid == AID_SHELL)
    allow(shell, orig_umask);
```

- Looks ok now, but sketchy code in the past:

```
@@ -318,7 +318,8 @@ int main(int argc, char *argv[])
             }
         } else if (!strcmp(argv[i], "-s") || !strcmp(argv[i], "--shell")) {
             if (++i < argc) {
-                strcpy(shell, argv[i]);
+                strncpy(shell, argv[i], sizeof(shell));
+                shell[sizeof(shell) - 1] = 0;
             } else {
                 usage();
             }
```

Pros and Cons of Su/Superuser

- If USB debugging enabled, no root exploit needed to obtain all data
  - Grants root access to “shell” without prompt

- Enables self-administration
  - Can patch your own services
  - Can detect malicious activity more easily

- Introduces additional attack surface via potential vulnerabilities and presence of accessible setuid apps

Final Words

Final Words

- Impossible to evaluate "Android" security, especially data protection, without considering chipset and handset hardware

- Use disk encryption if it's available!

- Disable USB debugging access when not in use

- Rooting/modding is a double-edged sword
  - Allows manual patching of vulns, but may introduce additional vulns or exposures

Thanks To…

- [mbm]
- kmdm
- IEF
- Matt Mastracci
- Joshua Wise
- ShabbyPenguin
- k0nane
- jcase
- PlayfulGod

References

- http://tjworld.net/wiki/Android/HTC/Vision/BootProcess
- http://wiki.opticaldelusion.org/wiki/Motoactv
- http://www.droid-developers.org/wiki

Questions?

E-mail: drosenberg@vsecurity.com
Twitter: @djrbliss

Company:
http://www.vsecurity.com

Personal:
http://www.vulnfactory.org