Reversing and Auditing Android's Proprietary Bits

Joshua J. Drake

REcon 2013

June 23rd 2013

Agenda

- Introduction
- Background
- Proprietary Code
- Reversing
- Auditing
- Case Studies
- Conclusion / Q & A

Introduction

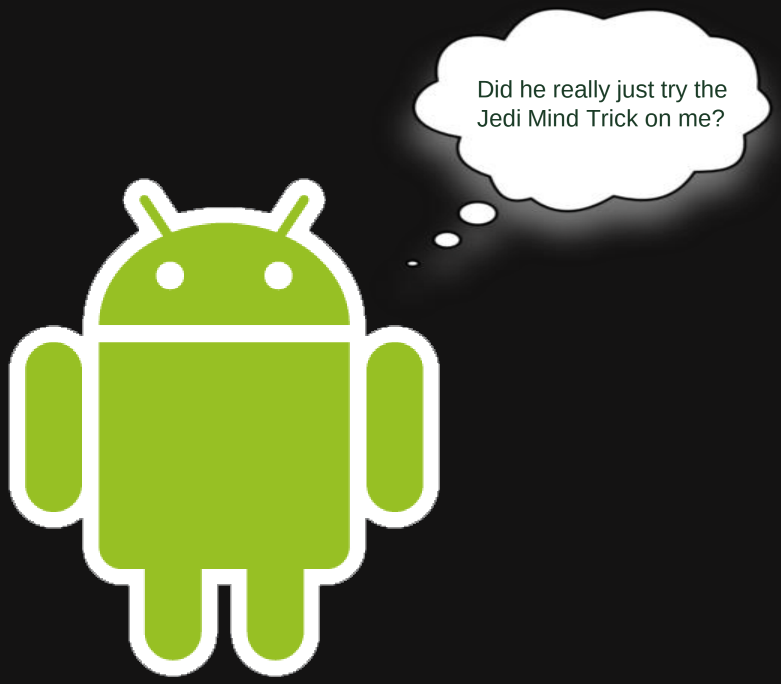
- Joshua J. Drake, aka jduck
  - Director of Research and Development
  - Previously Senior Research Consultant
  - Former Lead Exploit Developer at metasploit®
  - Research background:
    - Linux – 1994 to present
    - Android – 2009 to present
  - Demonstrated Android 4.0.1 browser exploit with Georg Wicherski at BlackHat USA 2012
  - Lead author of “Android Hacker’s Handbook”

BACKGROUND

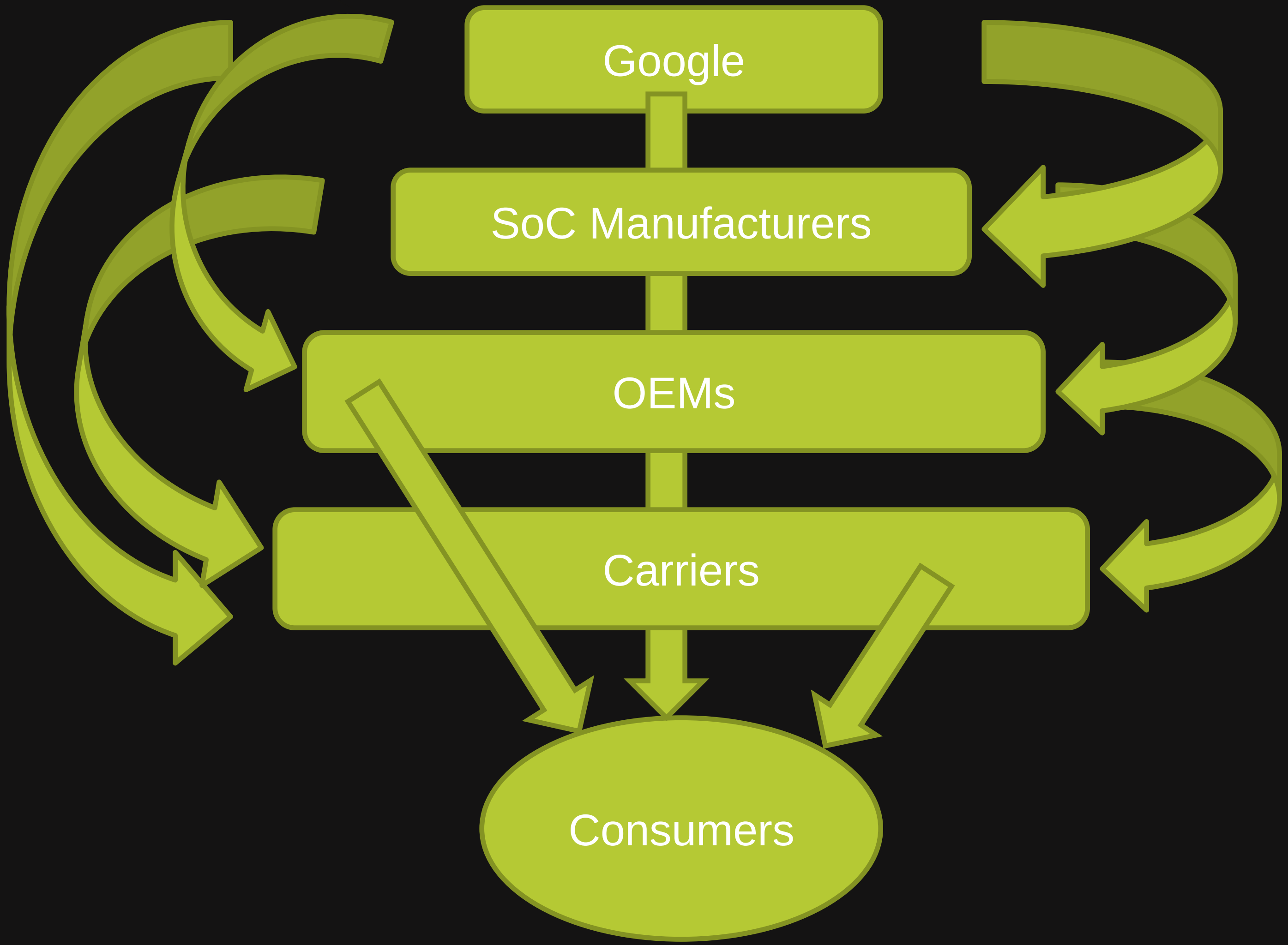
Why look at Android's proprietary bits?

Background – Android

- Android !!
  - Most common operating system (period)
  - Complex ecosystem
  - Primarily ARM devices
  - Linux based
  - “Open source”
  - Developed in Java/C/C++



Background – Ecosystem



Everything

Lower-level

Everything

Apps
boot loader
and radio
requirements

Diagram by Pau Oliva

Background - Devices

- Almost entirely ARM devices out there

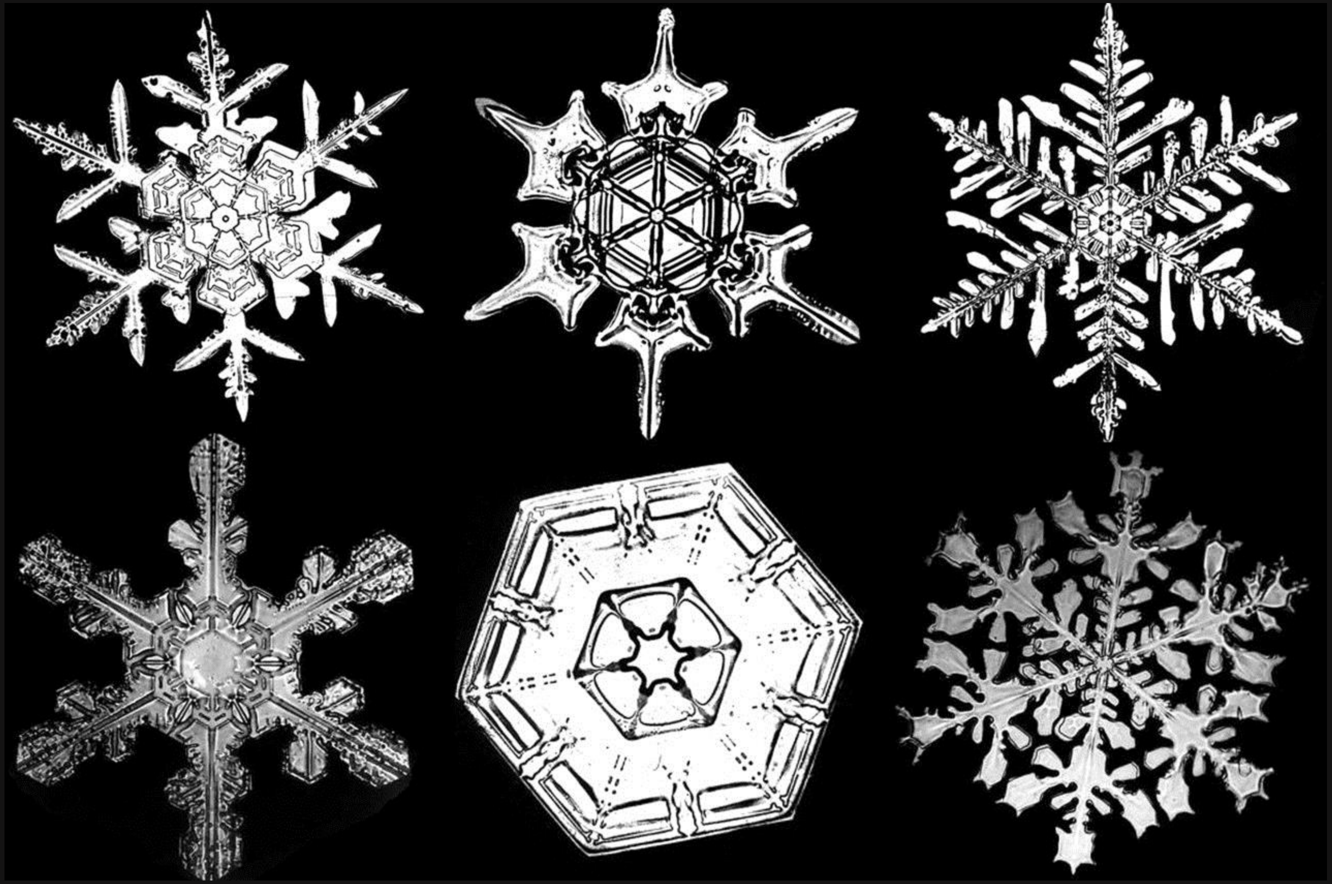


Image provided by Snowflake Bentley – http://snowflakebentley.com/

Background – My devices

Background – “Open source”

- Android Open Source Project (AOSP)

via Jean-Baptiste Quéru (AOSP maintainer, actually pushes the code)
https://plus.google.com/112218872649456413744/posts/g8YnZh5begQ

  - "Outside of proprietary device-specific files that come from hardware manufacturers, the basic rule is that everything is Open Source except the apps that talk to Google services: we want to be sure that the Android platform itself remains free of Google-specific code.”

  - Sums it up superbly!

Background – “Open source”

- Building your own firmware from AOSP requires binary blobs
- Nexus 4 factory images taken down shortly after they were first posted
  - Licensing issue maybe?
- Sometimes source code doesn’t match the bins
  - Example: Nexus 4 kernel config
    - live device has CONFIG_MODULES=n
    - Kernel source has CONFIG_MODULES=y

PROPRIETARY CODE

No source code, no docs, no bugs, right?

Proprietary Code – What kinds?

- Closed-source binary code is littered everywhere!
  - Third party licensed code
  - Nth party software
- You can find proprietary software…
  - In the kernel, modules
  - In user-space
  - In lower-level areas
  - Even apps
  - Really anywhere…

Proprietary Code – What kinds?

- Tons of stuff deep under the hood
  - Boot loaders
  - TrustZone OS / TZ apps
  - Baseband
- Kernel space drivers
  - Developed by OEMs or licensed from 3rd parties
  - File system drivers, WiFi, Bluetooth, etc
- User-space
  - rild / vendor-ril
  - TrustZone storage (no persistent storage in TZ)

Proprietary Code

- Device tree concept
  - Commonly heard in rom development communities
  - “device” directory in AOSP
  - Binary blobs required for a particular device
- Nexus devices
  - Nexus binary-only drivers page
- OEM devices
  - Only from “stock roms”, updates, live devices

Proprietary Code – Getting Bins

- Getting proprietary binaries is usually easy
- From “roms” or updates
  - Often requires special extraction methods
  - Google for “<device> stock rom”
  - Unpacking tools vary :-/
- From a live device
  - Dumping partitions
  - /vendor, /firmware, /sbin, other directories
  - Works even when no OTA or factory images are available!

Proprietary Code – Finding more

- Enumerating process list
  - Comparing it against a Nexus device
  - Exclude core services

```
root@android:/ # getprop ro.build.version.release
4.2.2
root@android:/ # ps | grep -v ' 2 ' | wc -l
56

root@cdma_maserati:/data # getprop ro.build.version.release
4.1.2
root@cdma_maserati:/data # ps | grep -v ' 2 ' | wc -l
79

ps | grep -v ' 2 ' | grep -Ev
'/(vold|rild|debuggerd|drmserver|mediaserver|surfaceflinger|installd|netd|
keystore|ueventd|init|servicemanager|adbd)'
```

Proprietary Code – Finding more

- Enumerating the file system
  - Again, diff against a Nexus device
  - Various directories to look in…
    - /vendor, /system/vendor
    - /firmware
    - /system/lib includes some too
    - Inside apps' data directories

  - As easy as a few shell commands

Proprietary Code – Finding more

- Enumerating the file system

```
dq:0:~/android/cluster$ ./oneliner.rb getprop ro.build.fingerprint | grep JRO
[*]   nexus-s: google/sojus/crespo4g:4.1.1/JRO03R/438695:user/release-keys
[*]      sgs3: samsung/d2spr/d2spr:4.1.1/JRO03L/L710VPBLJ7:user/release-keys
dq:0:~/android/cluster$ ./cmd.rb <DEVICE> su –c /data/local/tmp/busybox find / -print > /data/local/tmp/find.log
dq:0:~/android/cluster$ ls -l *.log
-rw------- 1 jdrake jdrake 4.2M Jun 23 12:18 nexus-s_find.log
-rw------- 1 jdrake jdrake 9.3M Jun 23 12:20 sgs3_find.log
dq:0:~/android/cluster$ grep ^/system/lib/ nexus-s_find.log | sort > 1
dq:0:~/android/cluster$ grep ^/system/lib/ sgs3_find.log | sort > 2
dq:0:~/android/cluster$ wc –l 1 2
   206 1
   539 2
```

Proprietary Code – Finding more

```
dq:0:~/android/cluster$ diff –ub 1 2
--- 1   Sun Jun 23 12:29:01 2013
+++ 2   Sun Jun 23 12:28:50 2013
@@ -4,49 +4,133 @@
 /system/lib/bluez-plugin/input.so
 /system/lib/bluez-plugin/network.so
 /system/lib/drm
+/system/lib/drm/libdivxplugin.so
+/system/lib/drm/libdrmwvmplugin.so
 /system/lib/drm/libfwdlockengine.so
+/system/lib/drm/libomaplugin.so
+/system/lib/drm/libplayreadyplugin.so
+/system/lib/drm/libtzprplugin.so
 /system/lib/egl
 /system/lib/egl/egl.cfg
+/system/lib/egl/eglsubAndroid.so
…and more…
```

Proprietary Code – Final note

- Make sure you look for the source first!
- Even though something looks closed, it may be based on open-source code
- Check and double-check
  - Source will save you time
  - If you still use the bins, the source can help lots

REVERSING

Source code is overrated.

Reversing

- Two approaches to reverse engineering
  - Static analysis
  - Dynamic analysis
  - There's real power in combining the two!
    - ie. resolving indirect code or data flow

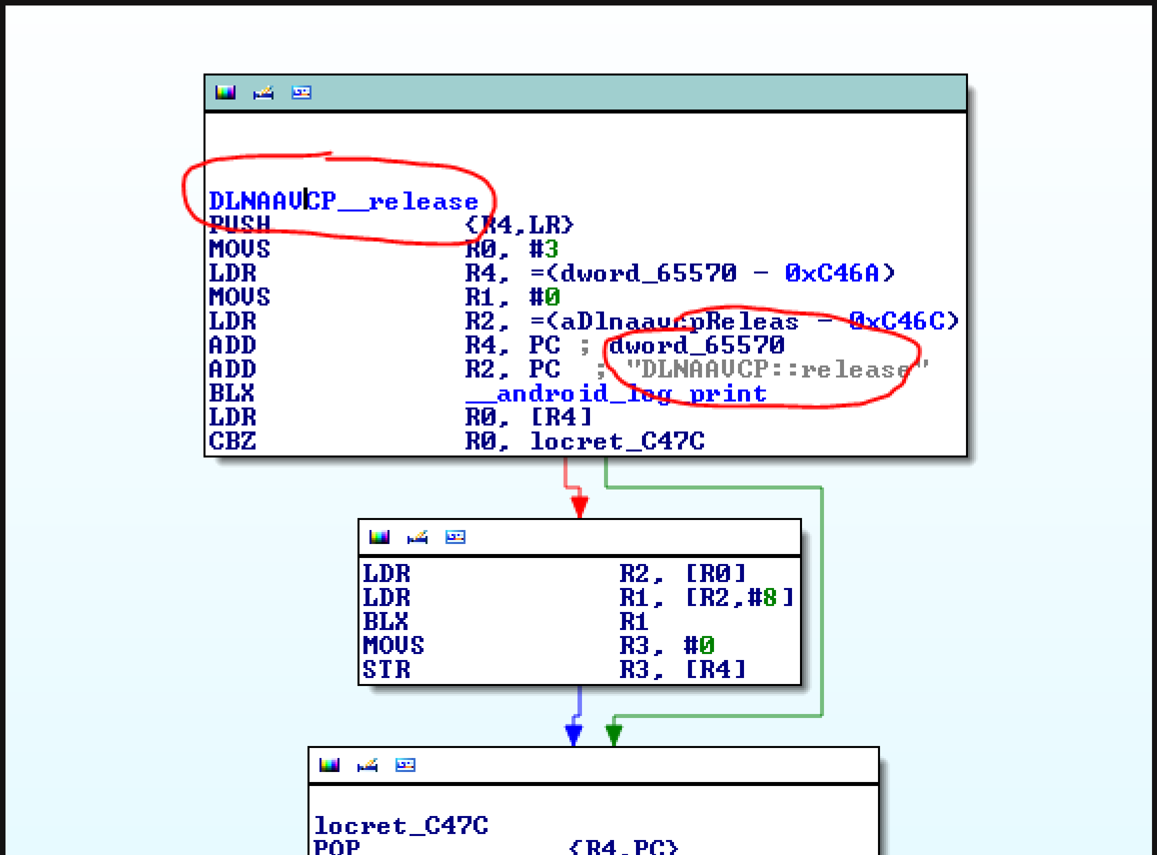
Reversing – Static Analysis

- Reversing ARM binaries can be tricky
  - Thumb vs ARM – troublesome and manual

  - Looking at ARMv7 bins with IDA Pro
    1. Open the binary
    2. Select ARM from processor type drop-down (tab, home)
    3. Click button “Set” button (tab, space)
    4. Click “Processor options” (alt-p)
    5. Click “Edit ARM architecture options” (tab, space)
    6. Click “ARM v7 A&R”
    7. Click “OK”, “OK”, “OK”
    8. Dig in!

Reversing – Static Analysis

- String analysis
  - Your best friend!



Reversing – Static Analysis

- De-compilation - Hex-rays helps!
  - Faster to read C-style pseudo code
  - Structure recovery
  - Type propagation
  - Great for C++
  - Some issues with Linux-kernel ASM functions

- Using symbols
  - Linux imports / exports are by name only
  - Common to find decent symbols

Reversing – Static Analysis

- Differential analysis

  - Comparing binaries
  - Comparing file system entries
  - Comparing running processes
  - Comparing specific files

  - Mostly for re-discovering known bugs
  - Useful for watching evolution of some code

Reversing – Static Analysis

- Grooming your IDB helps tremendously

- Look for:
  - Functions with tons of cross-references
  - Large functions

- If the bin has no imports (compiled static)
  - Try to identify common library functions first
    - memcpy, strcpy, strlen, strncpy, strlcpy, etc etc etc

Reversing – Dynamic Analysis

- User-space debugging options
  - logcat
    - Light on information, but still useful
    - Useful to see known strings
  - GDB
    - Apparently not the most stable tool
    - Python support in latest AOSP
    - Remote debugging is slow
    - Lack of symbols causes major problems

Reversing – Dynamic Analysis

- Symbols are more important on ARM

```
$ readelf –a binary | grep ‘ \$’
 31: 00008600     0 NOTYPE  LOCAL  DEFAULT    8 $a
 32: 000087b4     0 NOTYPE  LOCAL  DEFAULT    8 $d
  …
 37: 00008800     0 NOTYPE  LOCAL  DEFAULT    8 $t
```

  - $a – ARM code
  - $t – Thumb code
  - $d – Data

- GDB _relies_ on these
  - No symbols means manual ARM vs Thumb
  - Add 1 for Thumb when using x/i, setting breakpoints, etc
  - Use the thumb bit in $cpsr!

Reversing – Dynamic Analysis

- Instrumentation / Hooking
  - Much more efficient

  - Challenges
    - ARM vs Thumb (again)
    - Cache issues
    - No standard prologues
    - pc-relative data

  - Although tedious, can be achieved, see:
    - Collin Mulliner's android_dbi
    - saurik's Mobile Substrate

Reversing – Dynamic Analysis

- Kernel / boot loader debugging
  - JTAG (probably disabled)
  - USB-UART cables (Samsung and Nexus 4)
    - kgdb possible with a custom kernel
- Kernel debugging
  - proc file system (kmsg, last_kmsg)
  - Changing the kernel command line
    - Requires a custom boot.img

Reversing – Dynamic Analysis

- Instrumentation / Hooking
  - Again, much more efficient

  - kprobes, jprobes
    - Requires a custom kernel

  - Custom hooking
    - Needs only root
    - Same challenges as user-space

AUDITING

But didn't we fix all the bugs already?

Auditing

- Several methodologies
  - Top-down
    - Follows data flow / tainted input
  - API-based
    - Unsafe use of buffer functions
    - Format string vulnerabilities
    - Unsafe command execution usage
    - Checking memory allocations
  - Checking static buffer usage
  - Grep-for-bugs
    - Sign extension bugs
    - Integer overflows in allocations, etc

Auditing Tips

- Force Multipliers
  1. Learn as much as you can
  2. Deep understanding of the OS, APIs, architecture helps
  3. Taking advantage of source, docs, etc

- NO ASSUMPTIONS.

- Take lots of notes!
- Make comments and marks in IDA

Auditing – Binaries vs. Source

- Auditing binaries makes some bugs obvious
  - Pros
    - CPP macros are eliminated
    - Compiler may do something horribly wrong
    - No comments means no misleading statements
    - Likely to be less audited
  - Cons
    - More work to see the higher level
    - Binary auditing requires assembly skills
    - Unfortunately slower going
    - Dealing with indirection statically is a pain

Attack Surfaces – Low-level

- Low-level software attack surfaces
  - Boot loaders
    - partition table/data

Case Study - Loki

- Issue in the SGS4 boot loader
  - Discovered / released by Dan Rosenberg
  - For Qualcomm based devices (AT&T, VZW)
  - Allows bypassing secure boot chain
    - Can boot a custom kernel / ramdisk

- Samsung's "aboot", final stage boot loader
  - Verifies a signature on the "boot.img"
  - Based on the open source LK boot loader
  - Had a few modifications

Case Study - Loki

- Using the base source and binary from the device together helps get and stay oriented
- The code:

```
hdr = (struct boot_img_hdr *)buf;

  image_addr = target_get_scratch_address();
  kernel_actual = ROUND_TO_PAGE(hdr->kernel_size, page_mask);
…
  /* Load kernel */
  if (mmc_read(ptn + offset, (void *)hdr->kernel_addr, kernel_actual)) {
    dprintf(CRITICAL, "ERROR: Cannot read kernel image\n");
    return -1;
  }
```

Case Study - Loki

- OOPS!
- They trusted data in the boot.img header when reading from flash!
- Dan overwrote the aboot code itself
  - Replaced the signature checking function with his own
  - Simply fixed up the mess and returned success

Attack Surfaces – Low-level

- Low-level software attack surfaces
  - TrustZone
    - From ring0 only

Case Study - Motopocalypse

- Motorola TrustZone OS vulnerability
  - Discovered / released by Dan Rosenberg
  - Allows unlocking the boot loader
  - Could potentially allow more…

- Boot loader uses QFUSES
  - Can only be set one time!
  - Used by the OEM-supported unlock mechanism

Case Study - Motopocalypse

- TrustZone uses SMC instruction
  - Secure Monitor Call
  - Similar to how user-space calls kernel-space
  - Requires ring0 code execution
  - Processed inside TrustZone

- Dan found a bug in some TrustZone code

Case Study - Motopocalypse

- Inside Motorola's SMC handling code:

```
switch (code) {
    ...
    case 9:
      if ( arg1 == 0x10 ) {

        for (i = 0; i < 4; i++)
          *(unsigned long *)(arg2 + 4*i) = global_array[i];
        ret = 0;
      } else
        ret = -2020;
      break;
    ...
```

Case Study - Motopocalypse

- OOPS!
  - Attacker-controlled memory write!
- Dan overwrote an important flag
  - Enabled boot-loader-only SMC operations
  - Called OEM-supported unlock code
- Voila!
  - Unlocked boot loader via buggy proprietary code.

Attack Surfaces – Low-level

- Low-level software attack surfaces
  - Baseband
    - RF based attacks
    - From application processor

Case Study – S-OFF

- What is S-OFF?
  - “Security Off”
  - Relates to locked flash memory in HTC devices
  - Prevents writing to /system
    - Even with root
    - Event after remounting

- Some tools turn this off using baseband exploits!
  - They start with root, attack the baseband from the application processor

Attack Surfaces – Low-level

- Hardware attacks
  - USB – UART cables
    - Via headphone jack on Nexus 4
    - Using special OTG cable on Samsung devices
  - JTAG
    - Usually disabled
  - Other bus-based attacks
    - SPI
    - I2C
    - etc

Attack Surfaces – Kernel

- Custom / third party kernel modules

- Attack surfaces
  - Traditional Linux attack surfaces
  - proc, sys, debug, etc file systems
  - ioctl on open file descriptors
  - Custom implementations of POSIX apis
    - ie. custom mmap handler

  - Depends largely on the type of driver

Attack Surfaces – User-space

- Attack surfaces
  - Insecure file system permissions
    - Unsafe shell operations during boot
  - Socket endpoints (TCP, UDP, NETLINK, UNIX, abstract domain)
  - BroadcastReceivers, ContentProviders, etc
  - Enumerate via proc file system

UNDISCLOSED CASE STUDY

Oh, look! Bugs! Who knew?

Undisclosed Case Study

DEMO

CONCLUSIONS

Conclusions

- Fragmentation rampant
  - Complicates attacks
  - Helps defense a bit
- The ARM architecture is a PITA
- Proprietary bits of Android are great to audit
  - Requires more skills, less people have done it
- Buggy code, surely still more bugs lurking
- Donate unwanted Android devices to us!

PLEASE ASK QUESTIONS!

About Android, code, bugs, the book, anything…

Joshua J. Drake
Twitter: @jduck / IRC: jduck
jdrake [circled-a] accuvant.com
www.accuvant.com

ACCUVANT
THE AUTHORITATIVE SOURCE FOR INFORMATION SECURITY

Joshua J. Drake
Twitter: @jduck / IRC: jduck
jdrake [circled-a] accuvant.com
www.accuvant.com

BONUS SLIDES

These didn’t make the cut.

Background – "Open source"

- Android Open Source Project (AOSP)
  - Kind of a misnomer :-/

  - Google pushes their source after releases
    - Not true open source
    - Sets a bad example

  - Downstream (OEMs, etc) modify AOSP

- How many of you have checked out a copy?