A Totally Different View
Under the Hood with Android & Linux

Jonathan Levin
http://Technologeeks.com

Ready your Droids

• It helps to follow along and try some hands-on
  – (this is, after all, a tutorial, and not just another lecture)

• If you have a real device – great
  – But advanced tracing/debugging does need root access

• At a minimum, fire up a KK emulator, and adb to it.
What this isn’t

- An ADB shell primer
- A CLI how-to
- A native-level/NDK how-to
- A debugging primer for Dalvik/DDMS

What this is

- Collection of native and CLI level debugging techniques
- Uses AOSP-supplied tools, and Linux facilities
- Applicable primarily to ARM, but also Intel and MIPS
- Actually also usable for Linux native level debugging
- An excerpt from my upcoming Android Internals book
The Book

- “Android Internals: A Confectioner’s Cookbook”
- Unofficial sequel to “OS X and iOS Internals”
- To be released end of June, 2014
  - Along with Android Lollipop/Licorice/L* announcement?
- Updated for KK (4.4.2/API 19), with more to follow
- [http://newandroidbook.com/](http://newandroidbook.com/)
  - FAQ, TOC and plenty of bonus materials

Android Architecture

Mainstream Linux

- Applications (C/C++/…)
- Gnome/KDE/etc.
- X-Windows
- Native Libraries (glibc, …)
- Linux kernel
- Mostly Intel (but any) Hardware

Android:

- Applications (Java/DEX)
- JNI
- Frameworks
- Dalvik VM
- Native Libraries (Bionic, cutlfs, …)
- Hardware Abstraction Layer
- Linux kernel (2.6.21+ or 3.x) + Androidisms
- Mostly ARM (some Intel, MIPS) Hardware

<table>
<thead>
<tr>
<th></th>
<th>Mainstream Linux</th>
<th>Android:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native Binaries</td>
<td></td>
<td>Native Binaries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frameworks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dalvik VM</td>
</tr>
<tr>
<td>~75% same</td>
<td></td>
<td>Native Libraries (Bionic, cutlfs, …)</td>
</tr>
<tr>
<td>~95% same</td>
<td></td>
<td>Hardware Abstraction Layer</td>
</tr>
<tr>
<td>100% different</td>
<td></td>
<td>Linux kernel (2.6.21+ or 3.x) + Androidisms</td>
</tr>
</tbody>
</table>

~95% same

~75% same

100% different
ADB and the shell

• ADB provides a command line shell as uid shell

```
shell@htc_m8wl:/ $ id
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),1007(log),
1009(mount),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),
3002(net_bt),3003(inet),3006(net_bw_stats) context=u:r:shell:s0
```

• Shell is MirBSD Korn shell, with scripting abilities

• Recommendation: Install SSHD (or dropbear, etc)
  – Frees you from tethering requirement, fully remote
  – Allows easier (and safer) root access
  – Will require public key authentication only (no password..)

ToolBox vs. BusyBox

• Most CLI commands implemented via toolbox

• Toolbox is Android specific subset of busybox
  – Pros: linked with Bionic, recognizes AIDs
  – Cons: limited toolset, partial functionality

• Recommendation: Install BusyBox
  – Statically linked binary, so no dependencies
  – Can compile from source, but plenty of binaries out there
Getting tools to your device

- A lot of the tools you need are right in the emulator
  - Most in /system/xbin, and not present in most devices

- Can adb pull from emulator image, push to device

- Android version should match

- Remember to move libraries as well!
  - Find dependencies using objdump –x | grep NEEDED

The procfs (/proc) filesystem

- A plethora of diagnostic information:
  - General system diagnostics (/proc root)
  - Subsystem information (/proc/bus, /proc/irq, ...)
  - Sysctl variables (/proc/sys)
  - Per process diagnostics (/proc/[0-9]*))
The sysfs (/sys) filesystem

- Complements /proc, and provides:
  - Hardware and device representations
  - Kernel module information and parameters (/sys/module)
  - Kernel subsystem control

The debugfs (/d) filesystem

- Exclusively for kernel-level debugging and control
  - Kernel ftrace functionality
  - Binder debugging
  - Android atrace

- Usually mounted as /sys/kernel/debug, symlinked /d
keychords

- Little known feature of /init

- Binds services/commands to key combination
  - “keys” are physical buttons on device, as Android codes

- Uses /dev/keychord, where available

- Specify “keycodes” combination in /init.rc or other rc

Activity Diagnostics

- Tracing = monitoring run time activity of process

- Uses:
  - performance benchmarking
  - Logging and monitoring resource access
Activity Diagnostics - /proc

• A cornucopia of per process related information:

<table>
<thead>
<tr>
<th>/proc entry</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>/proc/$pid/cwd</td>
<td>Symbolic link to current working directory</td>
</tr>
<tr>
<td>/proc/$pid/cmdline</td>
<td>NULL separated argv[] of process</td>
</tr>
<tr>
<td>/proc/$pid/fd</td>
<td>Directory with symbolic links to open descriptors</td>
</tr>
<tr>
<td>/proc/$pid/fdinfo</td>
<td>Information about open descriptors</td>
</tr>
<tr>
<td>/proc/$pid/status</td>
<td>Human readable general statistics (VM + More)</td>
</tr>
<tr>
<td>/proc/$pid/task</td>
<td>Directory per thread</td>
</tr>
<tr>
<td>/proc/$pid/wchan</td>
<td>Wait channel (indicates kernel syscall block/sleep)</td>
</tr>
</tbody>
</table>

Activity Diagnostics - /proc

• Iterating over task/* will show threads

• Threads largely have same stats, save:
  – Status/Name – Dalvik threads are named with prctl(2)
  – wchan – Kernel wait channel/syscall

• You can grep name $t/status to isolate threads

• You can then trace or suspend specific threads
Activity Diagnostics - Tools

• AOSP provides the lsof tool to list open files
  – Not just files, but actually any file descriptor for process

• Extremely useful with grep to isolate files

Activity Diagnostics - Tools

• AOSP also provides the strace binary to trace syscalls
  – Hands down, the #1 debugging tool out there
  – Based on ptrace(2) API, no dependencies

• Useful in oh-so-may ways:
  – Can actually parse and present system call arguments
  – Can follow forks and threads
  – Can be used for timing of syscalls
  – Can introduce artificial latency(!)
Activity Diagnostics - Tools

- The ltrace tool can also be ported to Android
  - Similar to ltrace, but provides **library call** information
  - Uses ptrace(2), but a lot heavier, and needs libelf.

- Supplements strace when your problem is in a lib:
  - Arguments and features similar to strace
  - Can also be used for syscalls (with `–S`)

Activity Diagnostics - Triggers

- Linux doesn’t really support file access triggers
  - Inotify is an exception, but no shell command for it*
  - Still no notification for in-file access (say, certain offset)

- Using /proc and a little bit of scripting, however...

Memory Diagnostics

- RAM is the most important resource in Android
- Applications leave in perpetual fear of OOM/LMK
- Most memory in Android is shared when possible
- Important to understand memory diagnostics

Low Memory Killer

- Protector of all droids, sworn adversary of all apps
  - Linux OOM is not-so-deterministic.
  - LMK more predictable – but more conservative
- Each process has an oom_score and an oom_adj
  - Native apps can cheat death – Dalvik ones can’t
- LMK parameters can be tweaked through sysfs

<table>
<thead>
<tr>
<th>/sysfs entry</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>…adj</td>
<td>Array of oom_adj to kill on minfree hit</td>
</tr>
<tr>
<td>…minfree</td>
<td>Array of 4k multiples to start kill adj on</td>
</tr>
<tr>
<td>cost</td>
<td>Memory shrinker cost</td>
</tr>
<tr>
<td>debug_level</td>
<td>Verbosity for kill operations (1-5)</td>
</tr>
</tbody>
</table>
Memory Diagnostics

- **VSS**: Virtual Set Size (a.k.a VMSize)
- **RSS**: Resident Set Size
  
  \[ \text{USS} = \text{Unique Set Size} + \text{ShSS} = \text{Shared Set Size} \]
  
  \[ \text{HWM} = \text{RSS Peak} \]

- **PSS**: Proportional Set Size
  
  \[ \text{USS} + (\text{ShSS}/\#\text{Shares}) \]

Memory Diagnostics - /proc

- /proc filesystem provides key memory statistics:
  
  - Systemwide:

<table>
<thead>
<tr>
<th>proc entry</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>/proc/meminfo</td>
<td>Global memory statistics</td>
</tr>
<tr>
<td>/proc/vmstat</td>
<td>Kernel view of global memory, per variables</td>
</tr>
</tbody>
</table>

  - Per process:

<table>
<thead>
<tr>
<th>proc entry</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>/proc/$pid/maps</td>
<td>Address Space Layout (mmapped and anon)</td>
</tr>
<tr>
<td>/proc/$pid/smaps</td>
<td>As maps + per mapping information</td>
</tr>
<tr>
<td>/proc/$pid/status</td>
<td>VM Statistics, and much more</td>
</tr>
</tbody>
</table>
Memory Diagnostics - tools

- AOSP provides procrank and librank tools:
  - procrank: Ranks processes by memory utilization
  - librank: Ranks libraries by memory utilization (sharing)

- KitKat provides the memtrack tool
  - Logs memory utilization to Android logs

UpCall Scripts

- Android provides several CLI interfaces to Dalvik

<table>
<thead>
<tr>
<th>Upcall tool</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>am</td>
<td>Interface to Activity Manager</td>
</tr>
<tr>
<td>bmgr</td>
<td>Backup Manager</td>
</tr>
<tr>
<td>content</td>
<td>Interface with Android Content Providers</td>
</tr>
<tr>
<td>ime</td>
<td>Input-Method-Editors</td>
</tr>
<tr>
<td>input</td>
<td>Interface with InputManager, inject events, etc</td>
</tr>
<tr>
<td>monkey</td>
<td>Stress/Fuzz test tool</td>
</tr>
<tr>
<td>pm</td>
<td>Interface to PackageManager</td>
</tr>
<tr>
<td>settings</td>
<td>Get/set system settings</td>
</tr>
<tr>
<td>svc</td>
<td>Control power, data, wifi and USB</td>
</tr>
<tr>
<td>wm</td>
<td>Interact with the Window Manager</td>
</tr>
</tbody>
</table>
The service tool

- CLI interface to servicemanger and the Binder
- Simple, but powerful
- Can automate virtually all Android services
  – Does require root access for some of the services
- Depends on service called, virtually undocumented*


Input Events

- Toolbox’s getevent/sendevevent can automate input
- The input upcall script can inject to input manager

![Diagram of the Android input stack](image-url)
Post Mortem Debugging

- Android doesn’t support core dumps by default
  - Storage space is limited, and cores can be pretty big
  - `ulimit -c 0` is set in `/init` (via `setrlimit`) and inherited

- Tombstones used instead of cores
  - Application crashes, `debuggerd` is notified
  - Checks if `debug.db.uid` property is set, to wait for `gdb`
  - Otherwise, engraves “tombstone” (crash report)

Tombstones

- `Debuggerd` uses Linux’s `ptrace(2)` API to:
  - Enumerate all threads
  - Get register state for each thread
  - Get Stack trace for all threads
  - Get stack and instruction pointer memory contents

- Tombstone data is highly architecture specific
If you *do* want cores..

- Set ulimit to unlimited for crashing process
  - If you actively set to 0, must be root to unset

- Modify `/proc/sys/kernel/core_pattern`
  - Specify filename, can use `%h, %e, %u, %p`
  - Can also specify pipe `(`) and command name
    - Command accepts core via stdin (e.g. HTC's dalvik_coredump.sh)