Notes regarding Android OS System Development and Security as of Android 2.3.4

Nathaniel Husted

August 19, 2011

Contents

1 Compiling the Android OS Platform

This section will focus on successfully compiling the Android OS Platform. This includes all aspects of the OS development as well as the SDK, NDK, and various other toolchains used for Android application development. It will focus on the Ubuntu Operating System Version 10.04.2. This is the version used by Android developers at this time, at least until the next Ubuntu LTS release.

1.1 Required Libraries

The latest Sun Java JDK will need to be installed from the Ubuntu partner repositories. For Gingerbread on the sun-java6-jdk will be used.

Use the following libraries with apt-get install:

git-core gnupg flex bison gperf build-essential zip curl zlib1g-dev libc6-dev lib32ncurses5-dev ia32-libs x11proto-core-dev libx11-dev lib32readline5-dev lib32z-dev libgl1-mesa-dev gcc-multilib g++-multilib libc6-i386 libc6-dev-i386 git-core gnupg flex bison gperf build-essential zip curl zlib1g-dev libc6-dev lib32ncurses5-dev ia32-libs x11proto-core-dev libx11-dev lib32readline5-dev lib32z-dev libgl1-mesa-dev

1.2 Obtaining the Android OS sources

The documentation for obtaining the android sources is here: `http://source.android.com/source/downloading.html`. The documentation does not mention that the default download will download the latest *unstable* branch of the Android OS platform. A list of current stable builds as of is cupcake, donut, eclair, froyo, gingerbread. Note that they are the same as the codenames for the latest releases of the Android mobile platforms. You can then issue the following command to shift branches "`repo init -u` `git://android.git.kernel.org/platform/manifest.git -b <NewBranchHere>`" before syncing.

1.3 Building the Platform

While outdated, the rest of google's documentation is adequate for an initial build. The documentation can be found at: `http://source.android.com/source/building.html`.

Commands provided by the build environment that are not mentioned in this document, but also useful include:

- *croot*: Changes directory to the top of the tree.
- *m*: Makes from the top of the tree.
- *mm*: Builds all of the modules in the current directory.
- *mmm*: Builds all of the modules in the supplied directories.
- *cgrep*: Greps on all local C/C++ files.
- *jgrep*: Greps on all local Java files.
- *mgrep*: Greps on all makefiles
- *resgrep*: Greps on all local res/*.xml files.
- *godir*: Go to the directory containing a file.
- *printconfig*: tells you what configuration you are currently building.
- *choosecombo*: A more fine-grained alternative to lunch.
- *make dataclean*: clean the staging area for your data partition (i.e. data used by the emulator).
- *make installclean*: Cleans data related to switching between build variants.
- *make clean*: Remove everything.
- *make module1 module2 snod*: Builds provided modules and then rebuilds the system image without following dependencies.

- *BUILD_TINY_ANDROID=true make*: Builds a minimal version of the Android OS useful for kernel testing, debugger included.

1.4 Notes on the Android Kernel

The android kernel's used when building the platform are pre-built binaries that are downloaded with the Repo. To build a custom kernel variant for Android please refer to Section 2. To see what changes where made by Google in creating the Android kernel refer to Section 2.7.

1.5 Notes On Building For The x86 Platform

There is currently an Android-x86 project[1] whose goal is to port Android to the x86 platform in a fully working manner. This is a recommended way of obtaining a functional x86 version of Android as any required code modifications have already been done. A comprehensive set of build instructions can be found on their website. Much of the information for building the ARM versions of android apply.

Notes: The Gingerbread source tree had a number of errors out of the box. The Froyo tree compiled fine following their instructions. In order to run in VirtualBox, disable Mouse Integration.

1.6 Installing Android build on the Nexus One

The steps must be done in this *exact* order.

1. Run the build/envsetup.sh script
2. Run the device/htc/passion/extract-files.sh
3. Run "lunch full_passion-userdebug"
4. Run make
5. Reboot the phone into fastboot mode ("adb reboot fastboot" or hold the volume button down while turning the phone on).
6. Type 'fastboot -w flashall'

[1] `http://www.android-x86.org/`

2 Compiling the Android OS Kernel

The parimary target for compiling the Android kernel is the ARM platform. This is what a number of the Android OS builds default to. It is possible to build for the x86 platform as well, however that requires being able to build the Android OS platform for x86, a non-trivial task.

2.1 Obtaining the Android Kernel Source

The Android project contains a number of kernel varients in their GIT repository. These varients include:

- kernel/common - Comman Android Tree
- kernel/experimental
- kernel/linux-2.6 - Mirror of Linus's kernel sources
- kernel/lk
- kernel/msm
- kernel/omap
- kernel/qemu - Branch for use with the simulator
- kernel/samsung
- kernel/tegra

To obtain the source run the following command: "`git clone git://android.git.kernel.org/kernel/qemu`" or another target kernel repo. To obtain version information about the branch downloaded run "`git branch -r`" in the downloaded git repo directory. **NOTE:** If the kernel will be used with Android Emulator you *must* use a kernel with Goldfish in the branch name. The following was posted on the official build list regarding Goldfish.

> The Android emulator runs a virtual CPU that Google calls Goldfish. Goldfish executes ARM926T instructions and has hooks for input and output – such as reading key presses from or displaying video output in the emulator. These interfaces are implemented in files specific to the Goldfish emulator and will not be compiled into a kernel that runs on real devices.

To switch to a different kernel branch, the following two commands can be run:

- `git checkout -track -b <branch name> <branch location>`
- `git branch`

2.2 ARM Kernel Limiations

Linux does not officially support ARM SYSCALL auditing in its current form. However, a patch currently exists that enables auditing support in the ARM kernel. The files exist for both android-goldfish-2.6.29, linux-2.6.39.3, and linux-3.0 kernel versions.

2.3 Android MSM Kernel v2.6.35 Compilation Issues

The MSM kernel can be downloaded from the kernel/msm project on the Android GIT Repository site. The default configuration for the Nexus One is mahimahi_defconfig. It is also possible to pull the configuration file off a running nexus one.

The 2.6.36 branch of the Android kernel/common currently has two compile issues for the ARM platform. The first is that EM_ARM shows as undeclared. This can be fixed by adding 'include <asm/elf.h>' to the top of the 'arch/arm/kernel/ptrace.c' file. An include must also be added to <linux/audit.h> as well.

2.4 Obtaining Kernel Configurations

There are two ways in which a kernel configuration for the build can be obtained. The first is to run "`ARCH=arm CROSS_COMPILE=AndroidWorkingDirectory/-prebuilt/linux-x86/toolchain/arm-eabi-4.4.0/bin/arm-eabi- make menuconfig`" and manually add the required Android options stated in "Documentation/android.txt". The second is to issue the following command if the computer is connected to a phone (assuming the kernel option is enabled) or if the emulator is running: "`adb pull /proc/config.gz`".

2.5 Building The ARM Kernel

Building the kernel is quick after obtaining a configuration. The following command is all that is needed: "`ARCH=arm CROSS_COMPILE=AndroidWorkingDirectory/-prebuilt/linuxx86/toolchain/arm-eabi-4.4.0/bin/arm-eabi- make`". This method of cross compilation works with toolsets other than the one provided by Android. It has been tested to work with Code Sourcery Lite's[2] toolset as well.

2.6 Using the kernel

Kernel in the Emulator In order to use the newly built kernel the following argument can be used with the emulated "-kernel <PathToKernel>/zImage". Options can be passed to the kernel using the "-qemu -append options=values" command line argument.

Kernel in a Phone The kernel is installed as part of the boot.img file loaded with fastboot. In order to modify the kernel boot parameters the " file must be modified.

2.7 Differences in Kernel Sources

Linux For Devices did a thorough examination of Kernel changes by Google to the mainline Kernel so that it runs properly with Android devices[3]. Modifications were made to support the Goldfish platform (i.e. Google's android emulator), the YAFFS2 file system that runs on the phone, Bluetooth bug fixes, Scheduler modifications, power management, and miscellaneous changes to various subsystems (mainly additions). The major additional functionality Google added to the kernel was support for the IPC Binder, and Low Memory Killer, the Ashmem shared memory system, a RAM console and log device, and support for the Android Debug Bridge An annotated Kernel diff can be found on the Linux For Devices site as well[4].

[2] `http://www.codesourcery.com/sgpp/lite/arm/portal/subscription?@template=lite`

[3] `http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Porting-Android-to-a-new-device/`

[4] `http://www.linuxfordevices.com/files/misc/porting-android-to-a-new-device-p3.html`

3 Porting *nix Applications To Android

While Android does have a linux kernel and is based on the *nix platform, porting applications to the Android platform is a non-trivial task. This section will explain how to build new native applications into the Android Platform itself. Documentation for building native applications using the Android NDK can be found at `http://developer.android.com/sdk/ndk/index.html`. Section 4 includes more specifies on the differences between Android's bionic libc and Linux's GNU Libc.

3.1 Updating the build environment

First create a directory for the application in the "<WorkingFolder>/external" directory. Do not include the version of the application or anything besides the name of the application itself. If it is a shared library, then the standard convention also applies. The "external" directory includes a number of ported applications such as quake, strace, iptables, and bzip2. Make sure all the application's sources are in its directory.

3.2 Creating the Make files

Android uses its own build environment and also has its own makefiles, Android.mk. These files must in every directory that has a binary or library that is to be built. The basic structure of Android.mk files and a description of their various attributes are included in the Android NDK documentation. A Number of useful entries are:

- LOCAL_MODULE - The name of the local application or library
- LOCAL_MODULE_TAGS - The Android Build environment you want this application build in to
- LOCAL_SRC_FILES - The *.c or *.cpp files required to build the application or library
- LOCAL_C_INCLUDES - The location of every directory with includes
- LOCAL_CFLAGS - The CFLAGs to pass to the compiler

- LOCAL_LDLIBS - The compiled/linked libraries needed by the linker
- LOCAL_PRELINK_MODULE - true/false variable needed when building shared libraries in the project source environment (must be set to false)
- PRODUCT_COPY_FILES - <local file>:<file on target> - Places a local file in the target directory in the build. Useful for including default configuration files as all system directories in the emulator are read only.

3.3 Building the ported application

When all Android.mk files are added and correct, the application will be built as part of the general Android build process.

3.4 Notes on Kernel Header Locations

The android platform is confusing in that it has two directories for Linux kernel headers, the "external" directory and the "bionic/libc/kernel/common" directory. The "external" directory is not the headers that should be linked to and an open question remains of why there is a kernel-headers directory there. The "bionic/libc/kernel/common" directory is the official directory for the Linux kernel headers. The problem is, however, that this directory does not include all of the Linux kernel headers included with the kernel source itself so any downloaded kernel directory might be required to make up for these extra headers. One example is the "linux/audit.h" header that is included in official kernel packages but not in the "bionic/libc/kernel/common" directory. ARM specific headers are included in "bionic/kernel/arch-arm".

Documentation included specificaly states that the bionic headers are "clean" linux headers for inclusion into userland applications. These are created through a number of scripts in the "bionic/libc/kernel/tools" directory. The "clean" headers only contain type and macro definitions. A list of directories and their explanations follows, information originally in the "README.txt" in the project.

- *bionic/kernel/original* — "contains a set of kernel headers as normally found in the 'include' directory of a normal Linux kernel source tree.

note that this should only contain the files that are really needed by Android (use 'find_headers.py' to find these automatically)."

- *bionic/kernel/common* — "contains the non-arch-specific clean headers and directories (e.g. linux, asm-generic and mtd)"
- *bionic/kernel/arch-arm* — "contains the ARM-specific directory tree of clean headers."
- *bionic/kernel/arch-arm/asm* — "contains the real ARM-specific headers"
- *bionic/kernel/arch-x86* — "similarly contains all headers and symlinks to be used on x86"
- *bionic/kernel/tools* — "contains various Python and shell scripts used to manage and re-generate the headers"

Details on using the tools and the process in which they worked are detailed in the same "bionic/kernel/README.txt" file. The document also includes a diatribe on why the Android maintainers do this.

3.5 Android init.rc format

The init.rc file will be used by Android to start certain services and create certain system mountpoints.

A resource for the Android boot process can be found at `http://elinux.org/Android_Booting`. The "/init/readme.txt" is always a useful file.

4 Differences between Android Libc and Linux Libc

The Android Bionic libc (blibc) is significantly cut down compared to the GNU libc (glibc) used on Linux distributions. The blibc library is missing a number of headers, functions, and pre-processor macro definitions that are included in glibc. These differences can make porting applications difficult in the least, and near impossible in the worst.

It is specifically mentioned in the blibc documentation that Android does *not* support System V IPC. Google states the reason for not supporting these facilities is due to global kernel resource leakage. The header files effected are:

- <sys/sem.h>
- <sys/shm.h>
- <sys/msg.h>
- <sys/ipc.h>

Google also mentions that the blibc library is primarily a port of the BSD C library to the Android Linux kernel. It includes no support for locales, no support for wide characters, a small implementation of pthreads based on futexes, and support for x86, ARM, and ARM thumb CPU instruction sets/kernel interfaces. There is also no support for libthread_db or libm. To quote Google:

> Bionic doesn't want to implement all features of a traditional C library, we only add features to it as we need them, and we try to keep things as simple and small as possible. Our goal is not to support scaling to thousands of concurrent threads on multi-processors machines; we're running this on cell-phones, damnit !!

There has been debate within the linux community starting in 2010 on wether or not the Android OS was Linux or not Linux. The important point to note from this is that with later versions of the Linux kernel, all Android specific code was removed.

4.1 Alterantive Embedded gLibc's

While the Android group developed their own (bionic) libc for use with the Android platform, there are embedded (and compatible) versions to the glibc library. One example is the "eglibc" library [5] and uClibC[6].

[5] `http://www.eglibc.org/home`. Supported features can be found at `http://www.eglibc.org/cgi-bin/viewcvs.cgi/trunk/libc/option-groups.def?view=markup`.

[6] `http://www.uclibc.org/`

4.2 Cross Compilation vs. Android's Build Environment

There are two methods to compile ported applications for Android. The first is to use Android's built-in build environment. The second is to cross compile with a toolchain that supports ARM EABI. Both of these methods have caveats.

4.2.1 Cross Compilation Platforms

Scratchbox 1 This is the original Scratchbox platform[7]. It provides a emulated virtual to enable the cross-compilation of programs for the ARM platform. When started it creates a chrooted area with a complete system so that any applications compiled and installed will think of themselves as running on the ARM platform. The downside to this program is that it requires root access to install and the ARM tool-chain released with the latest version does not properly copy over the files it needs. Also, after following the installation instructions provided, the environment still will not properly compile/run ARM applications.

Scratchbox 2 This is a new version of scratchbox available with Ubuntu. It is similar to Scratchbox 1 in that it creates an emulated virtual environment, but operates from userland.

Code Sourcery Lite Toolchain This provies a set of tools precompiled to create ARM binaries on an X86 system.

4.3 Syscalls not in Android

Android and Linux syscalls differ as well. The list of Android syscalls can be found in "bionic/libc/SYSCALLS.TXT". Linux system calls were obtained by using the command "`man syscalls`". The following list of system calls were found in the linux man page but were not found when doing a search through "SYSCALLS.TXT". The functions are included below only if Android did not have the function in any form. For example Android contains "getegid32" but not "getegid". As no difference could be seen in the man pages, "getegid" was

[7] `http://www.scratchbox.org/`

not included in the list. This list includes roughly 152 system calls that are included in Linux and not in Android. When building the linux kernel from the 2.6.29 goldfish target, the build script specifically warns that fadvise64, migrate_pages, pselect6, ppoll, and epoll_pwait are not implemented.

_sysctl
accept4
add_key
adjtimex
alarm
alloc_hugepages
bdflush
break
create_module
dup3
epoll_create1
epoll_pwait
faccessat
fadvise64
fadvise64_64
fallocate
fchown
fgetxattr
flistxattr
free_hugepages
fremovexattr
ftime
futimesat
get_kernel_syms
get_mempolicy
get_robust_list
get_thread_area
getcpu
getpagesize
getsid
getxattr
gtty
idle
inotify_init1
io_cancel
io_destroy
io_getevents
io_setup
io_submit
ioperm
iopl
ipc
kexec_load
keyctl
lgetxattr
linkat
listxattr
llistxattr
lookup_dcookie
lremovexattr
lsetxattr
madvise1
mbind
migrate_pages
mknodat
mlockall
modify_ldt
move_pages
mpx
mq_getsetattr
mq_notify
mq_open
mq_timedreceive
mq_timedsend
mq_unlink
msgctl
msgget
msgrcv
msgsnd
munlockall
nfsservctl
nice
oldfstat
oldlstat
oldolduname
oldstat
olduname
pciconfig_iobase
pciconfig_read
pciconfig_write
personality
phys
pivot_root
ppoll
preadv
pselect6
putpmsg
pwritev
query_module
quotactl
readahead
readdir
readlinkat
recv
remap_file_pages
removexattr
request_key
restart_syscall
rt_sigpending

rt_sigqueueinfo
rt_sigreturn
rt_sigsuspend
sched_getaffinity
sched_setaffinity
semctl
semget
semop
semtimedop
send
set_mempolicy
set_robust_list
set_tid_address
setdomainname
setfsgid
setfsgid32
setfsuid
setfsuid32
sethostname
setxattr
sgetmask
shmat
shmctl
shmdt
shmget
signal
signalfd
signalfd4
sigreturn
splice
spu_create
spu_run
ssetmask
stime
swapoff
swapon
symlinkat
sync_file_range
sysfs
tee
tgkill
time
timerfd_create
timerfd_gettime
timerfd_settime
unshare
uselib
utime
utimensat
vhangup
vm86old
vmsplice

4.4 Functions not in Android Libc

The method for obtaining this list included running "`nm -D`" on "/lib/libc.so.6" and "/WORKING_DIR/out/debug/target/product/generic/obj/SHARED_LIBRARIES-/libc_intermediates/LINKED/libc.so". The output was grepped to only include symbols in the text (code) section ("T") and starting with a lower-case alphanumeric character (a-z). The two files were then diff'ed. There are roughly 587 functions included in the GNU libc environment that are not included in the Bionic libc environment.

a64l
abs
accept4
addseverity
argz_delete
atof
authdes_create
authdes_getucred
authdes_pk_create
authnone_create
authunix_create
authunix_create_default
callrpc
catclose
catgets
catopen
cbc_crypt
cfgetispeed

cfgetospeed
cfmakeraw
cfsetispeed
cfsetospeed
cfsetspeed
chflags
clearerr_unlocked
clnt_broadcast
clnt_create
clnt_pcreateerror
clnt_perrno
clnt_perror
clnt_spcreateerror
clnt_sperrno
clnt_sperror
clntraw_create
clnttcp_create
clntudp_bufcreate
clntudp_create
clntunix_create
confstr
create_module
ctermid
cuserid
des_setparity
dprintf
drand48_r
dup3
dysize
ecb_crypt
ecvt
ecvt_r
endaliasent
endfsent
endgrent
endhostent
endnetent
endnetgrent
endprotoent
endrpcent
endsgent
endspent
endttyent
endutxent
envz_add
envz_entry
envz_get
envz_merge
envz_remove
envz_strip
epoll_create1
epoll_pwait
ether_aton
ether_aton_r
ether_hostton
ether_line
ether_ntoa
ether_ntoa_r
ether_ntohost
faccessat
fallocate
fallocate64
fattach
fchflags
fcvt
fcvt_r
fdetach
feof_unlocked
ferror_unlocked
fexecve
fflush_unlocked
ffsll
fgetgrent
fgetpos64
fgetpwent
fgets_unlocked

fgetsgent
fgetspent
fgetws_unlocked
fgetxattr
flistxattr
fmemopen
fmtmsg
fopencookie
fputc_unlocked
fputs_unlocked
fputwc_unlocked
fputws_unlocked
fread_unlocked
freeifaddrs
fremovexattr
freopen64
fsetpos64
fsetxattr
fstatvfs
ftw
futimens
futimesat
fwrite_unlocked
gcvt
get_current_dir_name
get_kernel_syms
get_myaddress
getaliasbyname
getaliasbyname_r
getaliasent
getaliasent_r
getdate
getdirentries
getdomainname
getfsent
getfsfile
getfsspec
getgrent
getgrent_r
getgrgid_r
getgrnam_r
gethostbyaddr_r
gethostbyname2_r
gethostent_r
gethostid
getifaddrs
getipv4sourcefilter
getloadavg
getlogin_r
getmsg
getnetbyaddr_r
getnetbyname_r
getnetent
getnetent_r
getnetgrent
getnetname
getpass
getpmsg
getprotobyname_r
getprotobynumber_r
getprotoent
getprotoent_r
getpublickey
getpwent
getpwent_r
getpwnam_r
getpwuid_r
getrpcbyname
getrpcbyname_r
getrpcbynumber
getrpcbynumber_r
getrpcent
getrpcent_r
getrpcport
getsecretkey
getservbyname_r

getservbyport_r
getsgent
getsgent_r
getsgnam
getsgnam_r
getsid
getsourcefilter
getspent
getspent_r
getspnam
getspnam_r
getsubopt
getttyent
getttynam
getutmp
getutmpx
getutxent
getutxid
getutxline
getw
getwchar_unlocked
getwd
getxattr
glob
globfree
gnu_dev_major
gnu_dev_makedev
gnu_dev_minor
grantpt
gtty
hcreate
hcreate_r
hdestroy_r
host2netname
hsearch
hsearch_r
htonl
htons
iconv
iconv_close
iconv_open
if_freenameindex
if_nameindex
inet6_opt_append
inet6_opt_find
inet6_opt_finish
inet6_opt_get_val
inet6_opt_init
inet6_opt_next
inet6_opt_set_val
inet6_option_alloc
inet6_option_append
inet6_option_find
inet6_option_init
inet6_option_next
inet6_option_space
inet6_rth_add
inet6_rth_getaddr
inet6_rth_init
inet6_rth_reverse
inet6_rth_segments
inet6_rth_space
inet_lnaof
inet_makeaddr
inet_netof
inet_network
innetgr
inotify_init1
insque
ioperm
iopl
iruserok
iruserok_af
isastream
isfdtype
key_decryptsession

key_decryptsession_pk
key_encryptsession
key_encryptsession_pk
key_gendes
key_get_conv
key_secretkey_is_set
key_setnet
key_setsecret
l64a
labs
lchmod
lcong48
lfind
lgetxattr
linkat
listxattr
llabs
llistxattr
localeconv
lockf
lrand48_r
lremovexattr
lsearch
lsetxattr
lutimes
malloc_info
mblen
mbtowc
mcheck
mcheck_check_all
mcheck_pedantic
memfrob
mkfifo
mkfifoat
mkostemp
mkostemps
mkostemps64
mkstemps64
mlockall
mprobe
mrand48_r
mtrace
munlockall
muntrace
netname2host
netname2user
nfsservctl
nftw
nftw
nftw64
nftw64
nl_langinfo
ntp_gettime
ntp_gettimex
obstack_free
open_memstream
open_wmemstream
parse_printf_format
passwd2des
pivot_root
pmap_getmaps
pmap_getport
pmap_rmtcall
pmap_set
pmap_unset
posix_fadvise
posix_fadvise64
posix_fallocate
posix_fallocate64
posix_madvise
posix_spawn
posix_spawn_file_actions_addclose
posix_spawn_file_actions_adddup2
posix_spawn_file_actions_addopen
posix_spawn_file_actions_destroy
posix_spawn_file_actions_init

posix_spawnattr_destroy
posix_spawnattr_getflags
posix_spawnattr_getpgroup
posix_spawnattr_getschedparam
posix_spawnattr_getschedpolicy
posix_spawnattr_getsigdefault
posix_spawnattr_getsigmask
posix_spawnattr_init
posix_spawnattr_setflags
posix_spawnattr_setpgroup
posix_spawnattr_setschedparam
posix_spawnattr_setschedpolicy
posix_spawnattr_setsigdefault
posix_spawnattr_setsigmask
posix_spawnp
ppoll
preadv
preadv64
printf_size
printf_size_info
psiginfo
psignal
pthread_attr_getinheritsched
pthread_attr_setinheritsched
pthread_cond_broadcast
pthread_cond_destroy
pthread_cond_init
pthread_cond_signal
pthread_cond_timedwait
pthread_cond_wait
pthread_setcanceltype
putgrent
putmsg
putpmsg
putpwent
putsgent
putspent
pututxline
putwc_unlocked
putwchar_unlocked
pwritev
pwritev64
qecvt
qecvt_r
qfcvt
qfcvt_r
qgcvt
qsort_r
query_module
quick_exit
quotactl
rand
rand_r
rcmd
rcmd_af
readlinkat
realpath
recvmmsg
regexec
registerrpc
removexattr
remque
revoke
rexec
rexec_af
rpmatch
rresvport
rresvport_af
rtime
ruserok
ruserok_af
ruserpass
sched_getaffinity
sched_getaffinity
sched_getcpu
sched_setaffinity

sched_setaffinity
seekdir
semtimedop
setaliasent
setdomainname
setfsent
setfsgid
setfsuid
setgrent
sethostent
sethostid
sethostname
setipv4sourcefilter
setlogin
setnetent
setnetgrent
setprotoent
setpwent
setrpcent
setsgent
setsourcefilter
setspent
setttyent
setutxent
setxattr
sgetsgent
sgetspent
sigaddset
sigandset
sigdelset
sigemptyset
sigfillset
siggetmask
sighold
sigignore
sigisemptyset
sigismember
signalfd
sigorset
sigrelse
sigset
sigstack
sockatmark
splice
sstk
statvfs
stime
strerror_l
strfmon
strfry
strtof
strtold
stty
svc_exit
svc_getreq
svc_getreq_common
svc_getreq_poll
svc_getreqset
svc_register
svc_run
svc_sendreply
svc_unregister
svcerr_auth
svcerr_decode
svcerr_noproc
svcerr_noprog
svcerr_progvers
svcerr_systemerr
svcerr_weakauth
svcfd_create
svcraw_create
svctcp_create
svcudp_bufcreate
svcudp_create
svcudp_enablecache
svcunix_create

svcunixfd_create
swab
symlinkat
sync_file_range
tcflow
tcflush
tcgetsid
tcsendbreak
tcsetattr
tee
telldir
timegm
timerfd_create
timerfd_gettime
timerfd_settime
tmpnam_r
tr_break
ttyslot
ualarm
unshare
updwtmpx
uselib
user2netname
ustat
utimensat
utmpxname
versionsort
vhangup
vlimit
vmsplice
vswscanf
vtimes
vwscanf
wcstof
wcstoimax
wcstold
wcstoumax
wctomb
wordexp
wordfree
xdecrypt
xdr_accepted_reply
xdr_array
xdr_authdes_cred
xdr_authdes_verf
xdr_authunix_parms
xdr_bool
xdr_bytes
xdr_callhdr
xdr_callmsg
xdr_char
xdr_cryptkeyarg
xdr_cryptkeyarg2
xdr_cryptkeyres
xdr_des_block
xdr_double
xdr_enum
xdr_float
xdr_free
xdr_getcredres
xdr_hyper
xdr_int
xdr_int16_t
xdr_int32_t
xdr_int64_t
xdr_int8_t
xdr_key_netstarg
xdr_key_netstres
xdr_keybuf
xdr_keystatus
xdr_long
xdr_longlong_t
xdr_netnamestr
xdr_netobj
xdr_opaque
xdr_opaque_auth

xdr_pmap
xdr_pmaplist
xdr_pointer
xdr_quad_t
xdr_reference
xdr_rejected_reply
xdr_replymsg
xdr_rmtcall_args
xdr_rmtcallres
xdr_short
xdr_sizeof
xdr_string
xdr_u_char
xdr_u_hyper
xdr_u_int
xdr_u_long
xdr_u_longlong_t
xdr_u_quad_t
xdr_u_short
xdr_uint16_t
xdr_uint32_t
xdr_uint64_t
xdr_uint8_t
xdr_union
xdr_unixcred
xdr_vector
xdr_void
xdr_wrapstring
xdrmem_create
xdrrec_create
xdrrec_endofrecord
xdrrec_eof
xdrrec_skiprecord
xdrstdio_create
xencrypt
xprt_register
xprt_unregister

5 Android System Security

5.1 Binder, IPC, and VM Jailing

An overview of the security architecture and permissions can be found in `http://developer.android.com/guide/topics/security/security.html`. This subsection will contain a brief overview.

5.1.1 Binder and IPC

Android itself does not implement the traditional SysV IPC structure (See Sec. 4). It instead uses Binder as a method for allowing permissions based IPC between applications. Binder exists as both a Kernel driver and a userspace library. Work by Enck et al. makes brief mention of Binder describing it as a "component-based processing and IPC framework designed for BeOS, extended by Palm Inc., and customed for Android [1]." Binder serializes data objects into parcels and then passes the parcels between processes via a kernel module. The interface documentation for IBinder in the Google SDK provides more details on Binder's funcationality (`http://developer.android.com/reference/android/os/IBinder.html`).

5.1.2 Application Permissions

An Android application, by default, has no permissions at all, thus is can't access a number of Android OS system functionalities (e.g. the GPS unit). An application must request all the permissions it wants in the manifest file.

There are certain permissions assigned via the user/group system. These permissions can vary by manufacturer and are included in the /system/etc/permissions.xml file. A current application can check the groups it belongs to as well as its uid by running the "id" command. This command is available in both 2.2 and 2.3.4.

Applications can also define their own permissions in order to limit what other code can start its activities. User defined permissions have similar attributes in regrads to their permission group and protection level that android system permissions have. These permissions can be used to decalre who can start activities and services, who can send broadcasts to an associated receiver, who can access certain data in a content provider, and also who can read intents sent out by a context. The specific URI permission structure is in place to allow third party applications specific access to URI elements. The example given in the documentation is an email program sending an image attachment to an image view. The image viewer must have a way of receiving a read permission to that image so that it can be opened.

Applications that request too many permissions can have their permissions decreased upon reboot when the "/data/system/packages.xml" file is edited [8].

Each application is able to have read, write, and execute status to its own local directory at /data/data/<Package Name>. When an application is inserted here it is bound by some permissions [FIGURE OUT EXACTLY HOW THIS WORKS].

Ping Experiment The experiment used to test the permissions was performed by attempting to run a custom compiled (for the Android platform) ping binary. In this case the binary was the already compiled version that it housed in the <root>/external directory. The file was placed in the "as-

[8] `http://elinux.org/Android_Security`

set" directory of an Android project. The application, when installed, would then write the ping application from the "asset" directory to it's own personal /data/data/ directory. It then attempted to modify permissions on this file and run it locally. The application, however, gave the following error: "`socket: Permission denied`". This message changes to "`icmp open socket: Operation not permitted`" after the INTERNET permission is granted. The permission violations are caused by the ping application requesting to open a RAW socket. The ping application requires root access to perform this (the same goes for any other Linux kernel distribution). The ping application will then drop down to normal user privledges after opening the RAW socket.

Permission Checks Permission checks occur in the System API's. For example the WifiService.java file in "/frameworks/base/services/java/com/android/server" contains a number of references to the methods `enforceAccessPermission` and `enforceChangePermission`. These methods then refer to a method called `mContext.enforceCallingOrSelfPermission`. The android operating system maintains a list of all installed applications and their requested permissions in the "/data/system/packages.xml" file. There are known applications which will modify this file to prove a "user permission' firewall of sorts after applications have been installed[9].

A current test was performed attempting to run a "ping" application from this directory. It failed with a permission denied error. However, the system "ping" application worked. This is due to the system ping having the setuid bit set to run as root.

5.1.3 Applications Running In The Same Process

Two applications can run as the same user ID if they are signed by the same private key and request to be run in the same userID (`http://developer.android.com/guide/topics/manifest/manifest-element.html#uid`).

5.2 Application Installation and the Android Market

The android market is the primary method for installing applications on the Android phone system. Each application on the market is signed with a

9

developers private key. Some of the benefits of having all applications by the same developer signed with their own key are discussed in Section 5.1.3. The following site has details on what system changes occur during application installs: `http://benno.id.au/blog/2007/11/18/android-package`.

5.2.1 The Android Market

The application to access the Android Market from the Android platform is market.apk. Deviced must be certified compatible by Google before they receive official access to the market.apk and other official google .apk files. If they do not obtain the compatability certification or fail the certification, the only way to obtain these applications is via third party providers.

5.2.2 Installing non-Market APK files

This process is sometimes referred to as "sideloading". Some US cellular carriers restrict their users from installing applications that are not in the official Google Android Market. In order to install these applications a user had to follow a process of "sideloading" the applications. Sideloading involves plugging the phone into a computer via the USB cable and installing the application using Google's "adb" program. It is possible this process might required the phone to be "rooted", i.e. have the "su" program installed on the phone.

Some phones allow direct installation of applications on the phone. This feature is generally provided by third party roms (third party versions of android to install on mobile phones). This process generally requires a checkbox option to be selected on the phone that allows applications to be installed from third party sources.

6 APK File Structure and notes

The APK files that are distributed with the Android applications contain the DEX compiled java files as well as the resources the application uses. The APK files themselves are compressed files like JARs. The AndroidManifest.XML file contained therein, however, is repacked and no longer a text based XML file.

7 Android Escalation of Privledge Attacks

Two attacks against the Android platform currently exist. These attacks allow a user-level privledge to gain root privledges on the Android platform. They both exploit userland vulnerabilities in Android. Some of these exploits have been fixed, but the fixes depend on the manufacturer[10].

7.1 Rage Against The Cage

The Rage Against The Cage exploit takes advantage of a bug in the adb code. The adb code on android performs certain actions as root at start up but then drops root privledges using setuid. The exploit is in that the adb code does not check if the setuid call succeeds of fails. If the call fails adb will continue to run as the root user. Rage Against The Cage works by forking enough children processes to reach the NPROC limit on the machine and attempts to restart adb while NPROC is maxed. When this happens, the setuid call will fail in adb and it will continue to run with root privledges.

Details can be found at `http://intrepidusgroup.com/insight/2010/09/android-root-source-code-looking-at-the-c-skills/`.

7.2 Exploid

The exploid vulnerability takes advantage of the udev system on Android. The Google developers removed a large amount of code from udev as it would be implemented on Linux and moved the code into the init daemon. The dillema is that the udev code used is susceptible to a bug that existed in udev prior to 1.4.1 that did not verif that kernel messages it received came from the kernel. In the Android OS this means that init would receive these requests and init runs as root. A brief overview of the exploit is as follows:

1. Exploid copies it to a system directory writable to the shell user
2. It then sends a "NETLINK_KOBJECT_UEVENT" message to the kernel.
3. Copied executable checks to see if it srunning as root..
4. When running as root, remounts system partition as read-write

[10] http://c-skills.blogspot.com/2011/01/adb-trickery-again.html

5. Finally copies /system/bin/sh to /system/bin/rootshell and chmod's to 04711 to always run as root.

Details can be found at `http://intrepidusgroup.com/insight/2010/09/android-root-source-code-looking-at-the-c-skills/`.

7.3 KillingInTheNameOf

The KillingInTheNameOf exploit is slightly different in that it takes advantage of google's custom shmem interface "ashmem". The program maps the system properties into a processes adress space. The vulnerability is that they are not maped as write protected. The vulnerability then finds the ro.secure property of adb and flips it. That allows any shell started by adb to run as root. Rough details can be found in `http://jon.oberheide.org/files/bsides11-dontrootrobots.pdf`.

7.4 ZimperLich

The ZimperLich follows the same structure as the Exploid vulnerability. There are two major differences, though. First, ZimperLich attacks the Zygote process on android and its lack of a check against the failure of setuid. The Zygote process is the parent process which all Dalvik jails are forked from. the other difference is that attacking Zygote does not require a shell with a uid so the ZimperLich attack can be run from an APK. The source code for ZimperLich can be found at `http://c-skills.blogspot.com/2011/02/zimperlich-sources.html`.

7.5 GingerBreak

The GingerBreak exploit works in a similar manner to the Exploid vulnerability. The difference is that in this case the exploit takes advantage of the "vold" daemon improperly trusting messages recieved. A buffer underflow attack is commited that causes and escalation of privledge attack. The attack was found to work on on a number of devices from Android 2.2 to Android 3.0. The vulnerability is CVE-2011-1823. The very general description can be found at `https://groups.google.com/group/android-security-discuss/browse\_thread/thread/1ac1582b7307fc5c`. Source code for the exploit can be found at `http://c-skills.blogspot.com/2011/04/yummy-yummy-gingerbreak.html`.

This has supposedly been fixed in newer versions of the Android source [11] as of May 2nd.

8 ARM ABI vs. ARM EABI

The Android platform is built and runs on ARM EABI code. ARM introduced EABI to improve the floating point performance of their processors. It also modifies the system call convention for the ARM processor. This is import to know in certain activities where the Linux kernel must be modified. If the kernel is using the EABI instruction set, then the variables passed to the system call and the system call's number will be in different locations. The EABI supports 64-bit function parameters that are passed to even-numbered registers. The other major change is the system call number is passed to r7 whereas in ABI it was packed in with the base memory location.

Debians discussion of the port to EABI is discussed here: `http://wiki.debian.org/ArmEabiPort`. This is useful information for Ubuntu as well. The linux kernel handles both but the discrepancies are abstracted away when dealing with C-code in the kernel (except when accessing individual ARM registers).

9 Audit Linux Kernel Subsystem and Audit pacakage

The Audit Linux kernel subsystem and the Audit tool package for Linux work together to allow the monitoring of a Linux system. The audit package contains a number of programs that can be used.

- auditd – The daemon that interfaces with the audit kernel subsystem. If run with the -f option from a superuser command line, auditctl must be run first to load the rules.

- auditctl – A tool to add rules to the audit.rules file (this can be done manually as well). Also used when auditd is init'd to load all rules.

[11] `http://www.androidpolice.com/2011/05/03/google-patches-gingerbreak-exploit-but-dont-worry-we`

- audispd – A tool that is able to provide a userland daemon access to processed output from auditd.
- aureport – A tool used to produce summary reports of the audit logs.

9.1 Installing Linux Auditing

On most modern distributions the kernel is already configured to enable auditing and syscall auditing. All that needs to be done is for the user to install the audit package. However, in systems where the kernel must be manually compiled, the following configuration options need to be enabled: `CONFIG_AUDIT, CONFIG_AUDITSYSCALL, CONFIG_INOTIFY` and optionally `CONFIG_SECURITY_SELINUX`.

9.2 Audit Kernel Subsystem

The system call tracing occurs in the `do_syscall_trace` function in "ptrace.c". This function is called via the "entry-common.S" assembly file. When entering a system call the kernel checks to see if the "TIF_SYSCALL_AUDIT" flag is set. If it is, it performs a special slow system call path and calls the `syscall_trace` function. This function then enters the code in "ptrace.c". When the kernel performs the same check, with slightly different logic, when it returns from a fork.

9.3 Audit Logging Process

The Auditd program interracts with the audit system in the kernel by connecting through a netlink socket. If the daemon is not present, the events are recorded to the syslog daemon with printk. When syscall auditing is enabled the system calls number and timestamp are recorded at entry. When the syscall is exited, more information regarding the syscall is filled in.

9.4 Auditd on Android

In order to run the port of auditd on android it must be run from a root shell. This means that it must be initiated at OS startup or run through the adb shell. The process for starting auditd with spade is the following:

1. Login in to phone via root shell

2. Start auditd with (auditd or auditd -n)

3. Start spade-audit

9.4.1 Compiling audit Userland

The userland audit program must be configured prior to being set-up to work with the Android build system. This is because the "make" command must be run under the lib directory to generate the proper table header files for use with the system. The following instructions should compile all needed files:

1. ./configure –with-debug –disable-gssapi-krb5 –with-armeb –without-libwrap

2. cd ./lib

3. make

9.4.2 The Porting Process

The audit system was not a straightforward port to Android. A number of modifications needed to be made to the audit source where the source used references to unavailable functions or definitions. All references to `pthread_cancel` were switched to `pthread_kill` for this reason. The checks in "auditd.c" regarding permissions of exectubles it starts were removed as it was unable to properly read the permissions. The audit library also did not create the proper lookup tables for the arm platform used by the Android emulator. The google emulator's machine name returns as "armv5tejl". This was added to the machinetab.h file which creates the machinetabs.h file during the "make" process. The machine name of each arm phone is slightly different and will need to be added for each device. A new thread flag was created, _TIF_SYSCALL_WORK, that is equal to (TIF_SYSCALL_TRACE | TIF_SYSCALL_AUDIT). This flag is checked whenever entering or exiting a system call (or fork) to check for tracing.

Two things created large bugs on the arm platform. The first was that setting one thread flag equal to another via #define's did not work. The new code path did not execute properly. The second was the comments for the "ip" register were backwards. When tracing, $ip = 0$ is an exit, $ip = 1$ is an entry.

Another dilemma is that auditd will not start in daemon mode from the Android init.rc file, however auditctl will run. Auditd will run in daemon mode if started from the command line, however.

The list of kernel files currently modified are. Locations are from the kernel source base directory:

1. init/Kconfig
2. arch/arm/kernel/ptrace.c
3. arch/arm/inclyde/asm/thread_info.h

9.5 Creating a Kernel Patch

1. Download the latest stable kernel source.
2. Create a copy of the kernel source directory and rename it so that it's clear it is the modified branch.
3. Create a patch using the following command: diff -uprN -X linux-3.0-vanilla/Documentation/dontdiff linux-3.0-vanilla linux-3.0-modified > arm-audit-patch
4. Check the patch for formatting issues with the following command: perl linux-3.0-vanilla/scripts/checkpatch.pl arm-audit-patch
5. Manually check patch for errors

10 Mobile Malware Detection Techniques

10.1 Software Based Attestation

11 Notes of FaceNiff Android APK

The FaceNiff application can be downloaded at `http://faceniff.ponury.net/`. The goal of the application is to sniff WiFi packets and obtain the sessions cookies that websites send over HTTP. FireSheep[12] could be considered to be the odler cousin of this application. What makes FaceNiff an

[12] `http://codebutler.com/firesheep`

interesting specimin is that it is able to provide low level system functionality not formely thought possible on the Android platform. It does so by wrapping a C-based application in an Android APK. The limitation is that this application must be run as root on a rooted phone. The application can not only sniff wireless traffic but can perform ARP spoofing. As the primarily functionality is performed in the C-code, the reverse engineering of the DEX code only provided the run-time wrapper for the program.

Much of the application interacts with the "/proc" virtual filesystem. The Java portion executes certain IP tables commands. [WHAT DO THEY DO?]. The C application accesses much of the network routing functionality such as /proc/net/route and /proc/net/arp.

The following comments are made based on a "string" dump of the C application. The application makes use of SOCK_DGRAM sockets for a portion of its operation. The ARP spoofing are potentially donew ith SOCK_PACKET or SOCK_RAW sockets. Packet capturing is done with the libpcap library and the application supports Promiscuous mode. Due to certain hints in the dump such as "../iconv/skeleton.c", "../iconv/loop.c", "glibc-ld.so.cache1", it is quite possible the author statically compiled in a version of glibc. Considering the binary is 726K, this is a possibility.

Note: The author has also offered to make a WiFi jamming application as well.[13]

12 Literature Review

12.1 A Window into Mobile Device Security

Symantec's paper describes a broad overview of security in the iOS and Android operating systems. The technical paper focuses on the corporate environment. Symantec focuses on six difference types of threats in its analysis: web-based and network-based attacks, malware, social engineering attacks, resource and service availability abuse, malicious and unintentional data loss, and attacks on the integrity of the device's data. Symantec considers the implementations of both iOS and Android to be based on five different security pillars: traditional access control, application provenance, encryption, isolation, and permissions-based access control.

[13] `http://faceniff.freeforums.org/wifi-jammer-t52.html`

Apple's iOS provides traditional access control in the form of password configuration and account lockouts. The application provenance mechanism in iOS is very strong. Apple requires a membership to obtain both development tools and access to the apple application store. The cost is considerable and the developer is required to provide proof of identity. Also, all app submissions to the application store go through a testing process that apple does not divulge publically [Ed. - Although they divulge what your app must do to pass []]. While circumventing this process is possible, the authentication requirements mean that it will be difficult for malware authors to cheaply create multiple accounts. Jailbreaking an iOS device removes many of these protection measures and there have been two computer worm attacks effecting jailbroken iOS devices.

iOS makes use of full device encryption in iOS 4 using AES-256. An additional layer of encryption is used for email, and other sensitive data items. The downside to this design choice is the decryption key must be kept in memory at all times. The decryption key can be ex-filtrated if an individual has physical access to a device and can jailbreak it.

iOS provides application sandboxing. Each process is run inside it's own user id and has minimal access to the system. Applications cannot even determine if another app is running on the device. Applications must receive specific user participation for making phone calls an sending SMS messages. The following actions require no specific privilege requests:

1. Internet access
2. Address book access
3. Devices unique iOS id
4. The device's phone number
5. Device's music, video, and photo files
6. The safari search history
7. The auto-completion history
8. Recently viewed Youtube videos
9. WiFi connection logs

10. The microphone and camera

The sandboxing functions limit the dangers of malware on the device and protect resource abuse in most situations. With this sandboxing structure comes permissions-based access control. The iOS permissions are coarse grained only providing the following permissions: access to the location data from the GPS, the ability to receive remote notification alerts, the ability to initiate an outgoing call, and the ability to send an outgoing sms or email.

iOS currently has 200 different vulnerabilities found since its initial release. A majority of these are low severity. Very few allow privilege escalation vulnerabilities that allow for an attacker to gain control of the device. Apple takes an average of 12 days to patch these vulnerabilities. The following malware are examples of thsoe found on the platform: Aurora Feint, Storm 8, iPhoneOS.Ikee, and iPhoneOS.Ikee.B

The Android platform uses a non-standard java platform to run applications on its phones. The security is provided by direct linux permissions as well as fine grained permissions implemented in its framework. Applications by default can only access the following resources: a list of applications on the devices and its programming logic, the SD card's data (read-only), and may launch other applications.

The permission based access control allows permissions to various subsystems in the Android OS. Symantec provides the following high level categories: networking, device identifiers, messaging, calendar and address book, multimedia and image files, external memory card access, global positioning system, telephony system, logs and browsing history, and task list.

Android's digital signing model is much less rigid as Apple's. The Android digital signing model ensures applications cannot be tampered with and the applications author can be identified. Google, however, does not make rigid checks of developer provided personal information and the fee is considerably less. There is less cost to malware authors creating fake accounts to deliberate their malware on the store.

Android currently does not offer full device encryption for its phones although data encryption capabilities are in Android 3.0 for tablets. The platform has been found to have 18 vulnerabilities since its initial release. These have been patched in an average of eight days. The problem is that many of these patches take significant time to make it to the consumers due to the involvement of cellular providers and manufacturers. This structure creates ethical conflicts in the quick patching of problems. For example, Google had to exploit an escalation-of-privilege attack to remove the Android.Rootcager vulnerability. Example malware on the Android system include: Android.Pjapps, Android.Geinimi, AndroidOS.FakePlayer, Android.Rootcager, and Android.Bgserv.

Mobile antivirus scanners don't exist for iOS devices and they are impossible to implement due to iOS's strict isolation model. Antivirus tools exist on the Android platform but provide no support about unknown malware. Symantec assumes traditional AV scanners on mobile phones will be replaced with cloud-enabled reputation-based protection.

12.2 Android Permissions Demystified

Felt et al.'s paper focuses on the fine-grained permission system in the Android OS. Specifically they focus on identifying applications that request permissions well beyond what are needed. They do so by analyzing the Java Framework used by developers for Android. Their self stated contributions are the following:

1. The development of STOWAWAY, a tool to detect overpriviledge. It's used to evaluate 940 applications.

2. Identify and quantify patterns of developer error.

3. Map out Android's access control policy.

There are three levels of permissions in the Android system: normal, those that protect API access; dangerous, those that allow for money expenditure and private information gathering; signature/system, these regular access to very dangerous or fundamental permissions such as deleting applications and backing up the phone. Permissions are obtained at install time when application developers request them. Permission enforcement is done in the API. The permissions not enforced by the framework are INTERNET, WRITE_EXTERNAL_STORAGE, and BLUETOOTH. Those permissions are enforced via filesystem groups and the Linux kernel.

The API testing was performed in three steps. The first was feedback-directed testing. The second was customizable test case generation. The third was manual verification. Feedback-directed testing involved the use of the Randoop utility (automated feedback-directed OO test generator for

Java). To cover limitations of Randoop, the manual test cases were generated.The map generated from the first two methadologies was then checked manually.

The authors counted 1,665 classes in the framework with 16,732 public and private methods. They obtained 85% coverage of the API through the API calls. The uncovered portion of the API was due to native calls. The authors did not think there were any new permission checks hidden in these files. During this process the authors found errors in the documentation. 1207 API calls have permission checks, 779 of those are normal APIs and 428 are proxy interfaces. 20 of the 134 Android-defined permissions are unused. A number of permissions also have hierarchical relations.

The STOWAWAY application parses DEX files and identified standard API calls and also deals with Java reflection. WebView are specifically detected as well for INTERNET permission and an APK's XML files are parsed for mentions of WRITE_EXTERNAL_STORAGE. Using STOWAWAY on 940 Android 2.2 applications, the authors found that 323 (35.8%) had unnecessary permissions. The manual training phase found 42.5% (17/40) applications as having unnecessary permissions. The authors provide a number of reasons for developer in error for requesting these permissions: permission name confusion, confusion with related methods, copy and paste errors, actions from deputies, and artifacts from testing.

12.3 A Study of Android Application Security

Enck et al. study the data misuse of applications in the Android market including dangerous functionality and vulnerabilities. In order to analyze applications in this manner they produce a Dalvik decompiler (ded) and analyze $\tilde{2}1$ million lines of code from the top 1,100 free Android applications.

References

[1] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of OSDI*, 2010.

Appendix

Useful Commands

- 'ls -lt /var/lib/dpkg/info/*.list' – List the installed packages ordered by time/date they were installed.
- 'mount -o rw,remount -t yaffs2 /dev/block/mtdblock3 /system' – Remount Android /system with read/write privileges
- 'cat /proc/mounts' – Get mounted drives