Memory Analysis of the Dalvik (Android) Virtual Machine

Andrew Case

Digital Forensics Solutions
Who Am I?

• Security Analyst at Digital Forensics Solutions
  – Also perform wide ranging forensics investigations
• Volatility Developer
• Former Blackhat and DFRWS speaker
Agenda

• What is Dalvik / why do we care?
• Brief overview of memory forensics
• Extracting allocated and historical data from Dalvik instances
• Target specific Android applications
What is Dalvik?

• Dalvik is the software VM for all Android applications
• Nearly identical to the Java Virtual Machine (JVM) [1]
• Open source, written in C / Java
Why do we care?

• Android-based phones leading in US mobile market
  – Which makes for many phones to investigate

• Memory forensics capabilities against Android applications have numerous uses/implications

• Entire forensics community (LEO, .gov, private firms) already urging development of such capabilities
Memory Forensics Introduction

• Memory forensics is vital to orderly recovery of runtime information
• Unstructured methods (strings, grep, etc) are brittle and only recover superficial info
• Structured methods allow for recovery of data structures, variables, and code from memory
• Previous work at operating system level led to recovery of processes, open files, network connections, etc [4,5]
Memory Analysis Process

• First, need to acquire memory
  – Acquisition depends on environment [6]

• Next, requires locating information in memory and interpreting it correctly
  – Also requires re-implementing functionality offline

• Then it needs to be displayed in a useful way to the investigator
Dalvik Memory Analysis
Acquiring Memory – Approach 1

• The normal method is to acquire a complete capture of physical RAM
• Works well when analyzing kernel data structures as their pages are not swapped out
• Allows for recovery of allocated and historical processes, open files, network connections, and so on
Approach 1 on Android

- Without /dev/mem support, need a LKM to read memory
- No current module works for Android (ARM)
- We developed our own (mostly by @jtsylve)
- Benefits of full capture:
  - Can target any process (including its mappings)
  - Can recover information from unmapped pages in processes
Acquiring Memory – Approach 2

• Memory can be acquired on a per-process basis
• Ensures that all pages of the process will be acquired
• Easiest to perform with memfetch[8]
  – After a few small changes, was statically compiled for ARM
• No unmapped pages will be recovered though
  – Heap and GC don’t munmap immediately
Analyzing C vs Java

• Most previous forensics research has had the “luxury” of analyzing C
  – Nearly 1:1 mapping of code/data to in-memory layout

• Declaration of a C “string”
  – char buffer[] = “Hello World”;

• Memory Layout (xxd)
  – 4865 6c6c 6f20 576f 726c 6400 Hello World.
A Dalvik String in Memory

• First, need the address of the “StringObject”
• Next, need the offsets of the “java/lang/String” value and byte offset members
• StringObject + value offset leads us to an “ArrayObject”
• ArrayObject + byte offset leads to an UTF-16 array of characters
• ... finally we have the string (in Unicode)
Now for the memory analysis...

- The real goal of the research was to be able to locate arbitrary class instances and fields in memory
- Other goals included replicating commonly used features of the Android debugging framework
Locating Data Structures

• The base of Dalvik loads as a shared library (libdvm.so)
• Contains global variables that we use to locate classes and other information
• Also contains the C structures needed to parse and gather evidence we need
Gathering libdvm’s Structures

1) Grab the shared library from the phone (adb)

2) Use Volatility’s `dwarfparse.py`:
   - Builds a profile of C structures along with members, types, and byte offsets
   - Records offsets of global variables

3) Example structure definition

   'ClassObject': [ 0xa0, {
       'obj': [0x0, ['Object']],
   } ]

   Class name and size
   member name, offset, and type
Volatility Plugin Sample

• Accessing structures is as simple as knowing the type and offset
  intval = obj.Object(“int”, offset=intOffset, ..)

• Volatility code to access ‘descriptor’ of an ‘Object’:
  o = obj.Object("Object", offset=objectAddress, ..)
  c = obj.Object("ClassObject", offset=o.clazz, ..)
  desc = linux_common.get_string(c.descriptor)
gDvm

• *gDvm* is a global structure of type *Dvm Globals*
• Holds info about a specific Dalvik instance
• Used to locate a number of structures needed for analysis
Locating Loaded Classes

• gDvm.loadedClasses is a hash table of ClassObjects for each loaded class
• Hash table is stored as an array
• Analysis code walks the backing array and handles active entries
  – Inactive entries are NULL or have a pointer value of 0xcbcaccddd
Information Per Class

• Type and (often) name of the source code file
• Information on backing *DexFile*
  – *DexFile* stores everything Dalvik cares about for a binary
• Data Fields
  – Static
  – Instance
• Methods
  – Name and Type
  – Location of Instructions
Static Fields

• Stored once per class (not instance)
• Pre-initialized if known
• Stored in an array with element type `StaticField`
• Leads directly to the value of the specific field
Instance Fields

• Per instance of a Class
• Fields are stored in an array of element type \textit{InstField}
• Offset of each field stored in \textit{byteOffset} member
  – Relative offset from \textit{ClassObject} structure
Listing Instance Members

Source file: ComposeMessageActivity.java
Class: Lcom/android/mms/ui/ComposeMessageActivity;

Instance Fields:
  name:   m_receiver
  signature:  Landroid/content/BroadcastReceiver;

  name:   m_filter
  signature:  Landroid/content/IntentFilter;

  name:   mAppContext
  signature:  Landroid/content/Context;

  name:   mAvailableDirPath
  signature: Ljava/lang/String;
Analyzing Methods

• We can enumerate all methods (direct, virtual) and retrieve names and instructions
• Not really applicable to this talk
• Can be extremely useful for malware analysis though
  – If .apk is no longer on disk or if code was changed at runtime
Methods in Memory vs on Disk

• Dalvik makes a number of runtime optimizations [1]

• Example: When class members are accessed \textit{(iget, iput)} the field table index is replaced with the direct byte offset

• Would likely need to undo some of the optimizations to get complete baksmali output
Analyzing Specific Applications
Recovery Approach

• Best approach seems to be locating data structures of UI screens
  – UI screens represented by uniform (single type) lists of displayed information
  – Data for many views are pre-loaded
Finding Data Structures

- Can save substantial time by using *adb’s logcat* (next slide)
  - Shows the classes and often methods involved in handling UI events
- Otherwise, need to examine source code
  - Some applications are open source
  - Others can be “decompiled” with baksmali [9]
logcat example

The following is a snippet of output when clicking on the text message view:

D/ConversationList(12520): onResume Start
D/ComposeMessageActivity(12520): onConatctInfoChange
D/RecipientList(12520): mFilterHandler not null
D/RecipientList(12520): get recipient: 0
D/RecipientList(12520): r.name: John Smith
D/RecipientList(12520): r.filter() return result
D/RecipientList(12520): indexOf(r)0
D/RecipientList(12520): prepare set, index/name: 0/John Smith
Phone Call History

• Call history view controlled through a `DialerContactCard$OnCard ClickListener`

• Each contact stored as a `DialerContactCard`

• Contains the name, number, convo length, and photo of contact
Per Contact Call History

• Can (sometimes) retrieve call history per-contact
• Requires the user to actually view a contact’s history before being populated
Text Messages

• Recovery through ComposeMessageActivity & TextMessageView
• Complete conversations can be recovered
• Not pre-populated
Voicemail

- Audio file is open()’ed
- Not mapped contiguously into the process address space
- No method to recover deleted voicemails.
Browser (Opera Mini)

- Opera Mini is the most used mobile browser
- Can recover some session information
  - The history file is always mapped in memory (including information from current session)
- HTTP requests and page information is (possibly) recoverable
  - Can recover `<title>` information
  - Stored in Opera Binary Markup Language
  - Not publicly documented?
Recovering Wireless Information

• Screenshot on the right shows results of a scan for wireless networks
• Recovery of this view provides the SSID, MAC address, and enc type for routers found
• Recovery of “Connected” routers show which were associated with
Other Wireless Information

• Potentially interesting information:
  – Wireless keys
  – Connection stats

• These are not controlled by Dalvik
  – Keys only initially entered through Dalvik, but then saved

• Stored by the usual Linux applications
  – wpa_supplicant, dhcpd, in-kernel stats
Location Recovery

• Associating location & time not always important
  – But makes for better slides *hint*

• Interesting for a number of reasons
  – Forensics & Privacy concerns
  – Not part of a “standard” forensics investigation
Google Maps

• Did not do source code analysis
  – Most phones won’t be using Google Maps while being seized
  – Wanted to find ways to get historical data cleanly

• Found two promising searches
  – mTime=$TIME$, mLatitude=$LAT$, mLLongitude=$LON$
  – point: $LAT,LON$ ... lastFix: $TIME$
    • $TIME$ is the last location, extra work needed to verify
“Popular” Weather Application

- The weather application uses your location to give you relevant information
More GPS Fun

- All of the following applications do not clear GPS data from memory, and all send their lat/lon using GET with HTTP
  - Urban Spoon
  - Weather Channel
  - WeatherBug
  - Yelp
  - Groupon
  - Movies
Implementation

• Recovery code written as Volatility [7] plugins
  – Most popular memory analysis framework
  – Has support for all Windows versions since XP and 2.6 Intel Linux
  – Now also supports ARM Linux/Android

• Makes rapid development of memory analysis capabilities simple

• Also can be used for analyzing other binary formats
Testing

• Tested against a HTC EVO 4G
  – No phone-specific features used in analysis
  – Only a few HTC-specific packages were analyzed

• Visually tested against other Dalvik versions
  – No drastic changes in core Dalvik functionality
Research Applications

• Memory forensics (obviously)
• Testing of privacy assurances
• Malware analysis
  – Can enumerate and recover methods and their instructions
Future Avenues of Research

• Numerous applications with potentially interesting information
  – Too much to manually dig through
  – Need automation
  – Baksmali/Volatility/logcat integration?

• Automated determination of interesting evidence across the whole system
  – Combing work done in [2] and [3]
Questions/Comments?

• andrew@digdeeply.com
• @attrc
References - 1


References - 2