Plan

1. Introduction
2. Forensics: context
3. Forensics: memory
4. Forensics: filesystem
5. Reverse engineering
6. Conclusion
A few words on Android

Software:
- Linux kernel (patched)
- custom userland code: utilities, Bionic libc (BSD licensed)
- Java applications running on the Dalvik VM
- native code via JNI
- apps are (mainly) distributed on the marketplace

Hardware:
- mostly ARM but also MIPS, x86, PPC
- now powering TVs, tablets, ebook readers, etc.

Security model:
- one UID per application for isolation
- permission model for applications (GPS, phone, data, ...)
- relies on the security of the Linux kernel
Applications: APK

APK content

classes.dex
AndroidManifest.xml
resources.arsc
lib/
lib/armeabi/
lib/armeabi/libhello-jni.so
META-INF/
META-INF/MANIFEST.MF
META-INF/CERT.RSA
META-INF/CERT.SF
res/
res/layout/
res/layout/main.xml
This talk

Covers:

▶ physical memory (RAM) acquisition and analysis
▶ filesystem acquisition and analysis
▶ application reverse engineering

Does not cover:

▶ user data forensics (SMS, emails, etc.), use existing tools
▶ device specific tricks: jailbreaking/rooting, etc.

Research to create the SSTIC challenge:

▶ French IT security conference
▶ included forensics, reverse and cryptography
▶ awesome solutions (in French, except one) online
Plan

1. Introduction
2. Forensics: context
3. Forensics: memory
4. Forensics: filesystem
5. Reverse engineering
6. Conclusion
Android is a loosely defined platform:

- Android is just an upstream distribution (like kernel.org for Linux)
- manufacturers and carriers can and do customize it
- hardware varies: CPU, GPU, accessories
- evolution is extremely fast: 5 major releases in 1.5 years

Rogue apps exist:

- Jon Oberheide PoC *RootStrap*
- applications leaking informations (see TaintDroid)

Forensics experts need be able to deal with all these factors
Got root?

The root issue:

- most phones have NO root access
- root access is needed to dump the RAM and filesystems
- most root exploits, if they exist, need a reboot
- trust the exploit? UniversalAndroot has 800K of ELF binaries
- a reboot means losing a lot of potentially interesting data

A broken model:

- carriers lock users out, are slow to push out updates
- old, unsupported versions still distributed
- bad guys can root your phone using unpatched vulnerabilities
- you should not have to use vulnerabilities yourself to check/fix your system!

The following assumes root access, an ideal situation
Plan

1. Introduction

2. Forensics: context

3. Forensics: memory

4. Forensics: filesystem

5. Reverse engineering

6. Conclusion
Memory: acquisition

Usual way on Linux:

- parse `/proc/iomem` to identify RAM mappings
- `dd` on `/dev/mem` if it’s present (no `STRICT_DEVMEM` on ARM)
- use a kernel module (like `fmem`) if `/dev/mem` doesn’t exist

It gets uglier:

- unfortunately, `/dev/mem` is not always present (HTC, Acer)
- kernel modules are version, `.config` and compiler dependent
- that’s easy (in theory): get the source!
  - is it available?
  - is it really the exact version?
  - even if the GPL mandates it, it’s not always perfect
- `.config`: `/proc/config.gz`, if it’s enabled!

In practice it can take hours for each model.
Memory: analysis, generic

Rather well documented for x86, most common tasks include:

1- rebuilding processes
2- identifying open files
3- recovering open sockets

Usual way:

▶ identify structure member offsets for the given kernel version
▶ find the *pid 0* task using it’s *comm* field (swapper)
▶ walk the linked list of processes
▶ use the *mm_struct* to rebuild the virtual address space
▶ parse VMAs to identify files

ARM is basically the same but ...
Memory: analysis, Android

Some specificities:

- RAM is not always mapped at address 0
- RAM may be split
- PAGE_OFFSET varies
- *kallsyms* seems to always be present
- no public tools (except SSTIC challenge solutions)

Promising research to apply: *kmem_cache*:

- used for fixed-size allocation in the kernel
- the SLAB allocator keeps more data than SLUB
- all phones seem to use the SLAB allocator
- useful for sockets, dead processes

But this is not the only way...
APK are just ZIP, why not carve them?

- ZIP has a lot of redundant metadata:
  - each packed file is described by a *local file header* (LFH)
  - at the end, several *central directory headers* (CDH) point to all previous LFH
  - finally, a *end of central directory record* (EOCDR) terminates the archive

- rebuilding:
  1. identify all EOCDR
  2. check if the first CDH is in the same page, if not, look for it
  3. collect the filename, sizes and CRC from each CDH
  4. find the matching LFH
ZIP file format

So far so good, but what about fragmentation?

- pages are 4096 bytes
- but ZIP streams are compressed and their entropy high
- the last page of a stream is followed by a LFH or a CDH

In practice:

- works only on small archives (exponential number of combinations)
- easier to implement than full memory analysis (no kernel dependancy)
- real world example: a few minutes to analyze a 96MB dump with a python implementation

One can also try to dump (small) dex files directly (magic number).
Plan

1. Introduction
2. Forensics: context
3. Forensics: memory
4. Forensics: filesystem
5. Reverse engineering
6. Conclusion
Prerequisites:

- root access is still required
- but rebooting should not be destructive

Two main acquisition techniques:

- use `dd` or `nanddump` to dump mtdblocks to the SD card
- use Nandroid to directly dump the files to the host computer

YAFFS2:

- log-based filesystem, designed for NAND
- use `yaffs2utils` or `unyaffs` to extract files
- data recovery should be investigated (wear leveling)
Most of the information is easy to get in `/data`:

- installed packages: `/data/system/packages.xml`
- `dalvik-cache` contains ODEX files
- `checkin.db` database contains info on connections
- application specific sqlite databases

Applications are installed in `/data/app`
Plan

1. Introduction
2. Forensics: context
3. Forensics: memory
4. Forensics: filesystem
5. Reverse engineering
6. Conclusion
Reverse: the dalvik VM

Dalvik:

- java bytecode is converted to dalvik opcodes (*classes.dex*)
- VM is register based instead of stack based
- native code is available *via* JNI (*Java Native Interface*)

Applications, APK:

- Dalvik code (*classes.dex*)
- native code (*.so*)
- ressources (images, interface, data)
- manifest and signature (app signing is mandatory, but self-signed accepted)
Reverse: disassembly, example source

Standard RC4 code in Java

```java
byte getbyte() {
    int x, y;
    byte sx, sy;

    x = (this.x + 1)&0xFF;
    sx = state[x];
    y = (sx + this.y)&0xFF;
    sy = state[y];
    this.x = x;
    this.y = y;
    state[y] = sx;
    state[x] = sy;

    return state[(sx + sy) & 0xff];
}
```
Reverse: disassembly: dexdump

Android SDK *dexdump* output

```
|000a98| com.anssi.secret.RC4.getbyte():B
|0000:  iget v4, v6, Lcom/anssi/secret/RC4;.x:I  // field@0011
|0002:  add-int/lit8 v4, v4, #int 1  // #01
|0004:  and-int/lit16 v2, v4,  #int 255 // #00ff
|0006:  iget-object v4, v6, Lcom/anssi/secret/RC4;.state:[B  // field@0010
|0008:  aget-byte v0, v4, v2
|000a:  iget v4, v6, Lcom/anssi/secret/RC4;.y:I  // field@0012
|000c:  add-int/2addr v4, v0
|000d:  and-int/lit16 v3, v4,  #int 255 // #00ff
|000f:  iget-object v4, v6, Lcom/anssi/secret/RC4;.state:[B  // field@0010
|0011:  aget-byte v1, v4, v3
|0013:  iput v2, v6, Lcom/anssi/secret/RC4;.x:I  // field@0011
|0015:  iput v3, v6, Lcom/anssi/secret/RC4;.y:I  // field@0012
|0017:  iget-object v4, v6, Lcom/anssi/secret/RC4;.state:[B  // field@0010
|0019:  aput-byte v0, v4, v2
|001b:  iget-object v4, v6, Lcom/anssi/secret/RC4;.state:[B  // field@0010
|001d:  aput-byte v1, v4, v2
|001f:  iget-object v4, v6, Lcom/anssi/secret/RC4;.state:[B  // field@0010
|0021:  add-int v5, v0, v1
|0023:  and-int/lit16 v5, v5,  #int 255 // #00ff
|0025:  aget-byte v4, v4, v5
|0027:  return v4
```
Reverse: disassembly: dexdump

Android SDK **dexdump** output

```java
com.anssi.secret.RC4.getbyte():B

iget v4, v6, Lcom/anssi/secret/RC4;.x:I
add-int/lit8 v4, v4, #int 1
and-int/lit16 v2, v4, #int 255
iget-object v4, v6, Lcom/anssi/secret/RC4;
  .state:[B
aget-byte v0, v4, v2
iget v4, v6, Lcom/anssi/secret/RC4;.y:I
```
Reverse: tools

\textit{baksmali/smali}:
- disassembler / assembler
- easier to use than \textit{dexdump}
- allows code modification and recompilation
- handles APK directly

\textit{android-apktool}:
- decodes/encodes resources
- includes smali/baksmali
- allows smali code debugging
Reverse: decompilation

Several tools convert the dex code back to standard Java bytecode:

- **undx** was the first one presented:
  - more a PoC: fails often
  - resulting code isn’t optimal
  - recent fork ig-undx may be better

- **dex2jar**:
  - mostly documented in Chinese
  - works quite well

Use jd-gui or Jad to decompile resulting jar
Reverse: decompilation

dex2jar output

```java
byte getbyte()
{
    int i = this.x + 1 & 0xFF;
    int j = this.state[i];
    int k = this.y + j & 0xFF;
    int m = this.state[k];
    this.x = i;
    this.y = k;
    this.state[k] = j;
    this.state[i] = m;
    byte[] arrayOfByte = this.state;
    int n = j + m & 0xFF;
    return arrayOfByte[n];
}
```
Reverse: ODEX

Dalvik special case: ODEX

- *Optimized DEX*
- platform-specific optimizations:
  - specific bytecode
  - vtables for methods
  - offsets for attributes
  - method inlining
- the code is way harder to read
- dex files recovered from memory are ODEX
- */data/dalvik-cache* contains ODEX
Reverse: ODEX

ODEX disassembly

|0000: +iget-quick v4, v6, [obj+000c]
|0045: +invoke-virtual-quick {v8}, [000d]

*baksmali* handles (deodex) ODEX code

- but needs all dependencies to resolve offsets
Reverse: JNI

JNI is the same in Dalvik as in Java:

- an external .so is loaded
- methods must use the JNIEnv structure to exchange information
- types must be converted to native type from Java types
- otherwise just native code: syscalls, libs, ...

JNI specificities can ease reversing (compared to standard C):

1- get the function signature in Java
2- use IDA to generate a TIL file from jni.h
3- assign the structure to the right variable
4- see function calls directly
5- do the same in Hex-Rays
Reverse: JNI

1.
```
MOV     R8,  R3
MOVS    R3,  0x2A4
LDR     R3, [R2,R3]
MOV     R9,  R1
MOVS    R2,  #0
LDR     R1, [SP,#0x1D0+var_1BC]
MOVS    R7,  R0
BLX     R3
```

2.
```
MOVS    R3,  0x139
LDR     R3, [R1,R0]
MOV     R9,  R1
MOVS    R2,  #0
LDR     R1, [SP,#0x1D0+var_1BC]
MOVS    R7,  R0
BLX     R3
```

3.
```
MOV     R8,  R3
MOVS    R3,  JNINativeInterface.GetStringUTFChars
LDR     R3, [R2,R3]
MOV     R9,  R1
MOVS    R2,  #0
LDR     R1, [SP,#0x1D0+var_1BC]
MOVS    R7,  R0
BLX     R3
```

Raphaël Rigo - ANSSI
Android: forensics and reverse engineering
Reverse: JNI with Hex-Rays

v13 = strlen(v23);
8j3zIX(&v25, v23, v13, 0);
8j3zIX(&v25, v36, 32, 0);
sd1Hj(&v25, &v32);
v22 = &v32;

((void *)__fastcall *)(JNIEnv *, int, __DWORD, signed int)(^jnienv_)->SetByteArrayRegion(jnienv_, v24, 0, 32);
i = 0;
do
    { v26[i] = v33[i] ^ v36[i & 7]; ++i; }
while (i != 17);
v27 = v26[0] ^ 0x31;
v28 = v26[1] ^ 0x2C;
v29 = v26[2] ^ 0x59;
v30 = v26[3] ^ 0x2F;
v15 = ((int *)__fastcall *)(JNIEnv *, unsigned __int8 *)(^jnienv_)->FindClass(jnienv_, v26);

(^jnienv_)->SetByteArrayRegion(jnienv_, v24, 0, 32);
The future

Needed:

- complete (reliable) tools for memory analysis
- advanced tools for filesystem analysis

The next steps:

- Android 2.2 introduced JIT, big deal?
- virtualisation (Acer Betouch E130)
- handle all kinds of CPUs
- more and more diversity to come
References

- http://communaute.sstic.org/ChallengeSSTIC2010
- http://viaforensics.com/wiki/
- http://www.illegalaccess.org/undx.html
- http://code.google.com/p/dex2jar/
- http://java.decompiler.free.fr/
Thanks and contact info

Thanks to: Cédric Bouhier, Arnaud Ebalard and the SSTIC challenge participants.

raphael.rigo (at) ssi.gouv.fr