EVOLUTION OF ANDROID EXPLOITS FROM A STATIC ANALYSIS TOOLS PERSPECTIVE

VB2014: Anna Szalay & Jagadeesh Chandraiah

Email {anna.szalay, jagadeesh.chandraiah}@sophos.com
## Worldwide Device Shipment by Operating System (Thousands of Units)


<table>
<thead>
<tr>
<th>Operating System</th>
<th>2013</th>
<th>2014</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>879,821</td>
<td>1,170,952</td>
<td>1,358,265</td>
</tr>
<tr>
<td>Windows</td>
<td>325,127</td>
<td>339,068</td>
<td>379,299</td>
</tr>
<tr>
<td>iOS/Mac OS</td>
<td>241,416</td>
<td>286,436</td>
<td>324,470</td>
</tr>
<tr>
<td>Others</td>
<td>873,194</td>
<td>683,519</td>
<td>565,186</td>
</tr>
<tr>
<td>Total</td>
<td>2,319,559</td>
<td>2,479,976</td>
<td>2,627,221</td>
</tr>
</tbody>
</table>
Malware exploits samples share
Exploits overview

Android Vulnerabilities Exploitation Timeline

- Q4 2010: Android WebKit browser exploit
- Q4 2010: Android Data Stealing Vulnerability
- Q1 2011: Android Local Root Exploit aka “Rage against the cage” or Lotoor exploit
- Q3 2011: Android ClientLogin Protocol Vulnerability
- Q3 2011: Android Gingerbreak root exploit
    
- 2012: DEX2JAR exploitation
- Q2 2013: “Master Key” vulnerability
- Q2 2013: “Extra Field” vulnerability
Exploits overview

Q1 2011

Android Local Root Exploit aka “Rage against the cage” or Lotoor exploit

- Andr/DroidRt aka DroidRoot (ELF EXECUTABLE)

- Andr/DroidD aka DroidDream

- Andr/KongFu aka DroidKungFu
Malware root exploit samples share
Share of root-exploit-based malware samples by threat

- DroidDream: 68%
- BaseBridge: 15%
- RootExploit/other: 14%
- DroidKungFu: 2%
- GingerMaster: 0.1%
Exploits overview

Q3 2011

Android Gingerbreak root exploit

- CVE-2011-1823

- allows local users to execute arbitrary code and gain root privileges via a negative index that bypasses a maximum-only signed integer check on the DirectVolume::handlePartitionAdded method, which triggers memory corruption

- Andr/Gmaster-A aka GingerMaster
Android Gingerbreak root exploit: Inside Gmaster APK, exploit code in a picture file, IDA image
Exploits overview

Q2 2013

DEX2JAR exploitation: Andr/Obad-A

- disrupts the conversion of Dalvik bytecode into Java bytecode
- exploits AndroidManifest.xml processing
- gains extended Device Admin privileges
complex code encryption

- all external methods are called via reflection
- all strings are encrypted, including the names of classes and methods
- each class has a local descriptor method which obtains the string required for encryption from the locally updated byte array
- all strings are hidden in this array
- additional stage of decryption for C&C addresses
Obad obfuscation, example of decompiled by dex2jar dex code output
Obad obfuscation, example of decompiled by dex2jar dex code output

```java
package com.android.system.admin;

public class oCIlC11 {
    private static final byte[] oIlclcIc = { 0, 4, -67,
    
    static {
        int i;
        if (!oCIlC11.class.desiredAssertionStatus())
            i = 1;
        else
            i = 0;
        oCIlC11 = i;
    }
```
Obad obfuscation, example of decompiled by dex2jar dex code output
Exploits overview

Q2 2013

‘Master Key’ vulnerability

- CVE-2013-4787

- reported to affect 99% of devices

- apk signature ‘compromised’

- allows to replace installed app with the compromised ‘update’
“Master Key” vulnerability: Andr/MstrKey-A

- Exploited apk file has more than one classes.dex and AndroidManifest.xml files.

- Dodgy manifest file shows permissions that won’t be seen when we look at its actual fingerprint.
Inside ‘Master Key’ malware, example of multiple files.
Exploits overview

Q3 2013

‘Extra Field’ vulnerability

- allows to bypass code verification

- uses an extra field during an APK verification as archive

- based on signed-unsigned integer mismatch

- exploits fundamentals of the Android APK as a ZIP archive with the some special object fields
Example of malware exploiting the ‘Extra Field’ vulnerability, showing a changed field
Milestones in exploiting...

- **Root Exploits**
  - Andr/DroidRt (DroidRoot)
  - Andr/DroidD (DroidDream)
  - Andr/Kongfu (DroidKungFu)
  - Andr/Gmaster (GingerMaster)

- **DEX2JAR**
  - Andr/Obad (Obad)

- **APK Signature**
  - Andr/MstrKey (MasterKey)
Evolution of Android Exploits from a Static Analysis Tools Perspective - A Szalay, J Chandraiah
What’s Next?

• Static Analysis Tools and Techniques
  • Discuss various Android Static Tools and Techniques

• Evaluation of Tools against Exploit Samples
  • Android Master Key Vulnerability
  • Dex Header
  • Unfamiliar Opcodes
  • Decompilation Issues
Why?

- To verify if tools really meet the requirements
- Highlight issues encountered by analysts
- Identify causes that break various tools
- Suggest best approach for improvement based on the outcome of the research
APK and key elements

- Classes.dex
- AndroidManifest.xml
- Resources.arsc
APK and key elements

Java → Class → Dex → Classes.dex

App Specific Data → Binary Xml → AndroidManifest.xml

Resource, Strings → Binary Xml → Resources.arsc
Static Analysis Techniques

Unarchiving

APK

Decoding
AndroidManifest.xml
Resources.arsc

Disassembling
classes.dex

Decompiling
Classes.dex
Static Analysis Tools

- **Unarchiving** – Unzip or any similar archive extraction tools.

- **Decoding** – Apktool, Androguard

- **Decompile** – Dex2jar, Jdgui, JEB, ded and many other java decompilers.

- **Disassembly** – smali/baksmali, dexdump and IDA pro
Methodology

• Most commonly used analysis technique and Tools used for testing.

• Selected group of popular and relevant Android Exploit samples were used for testing.
  - Andr/MstrKey, Andr/DroidD (DroidDream), Andr/DroidRt, Andr/Obad, Andr/Kongfu (DroidKungfu), Andr/Gmaster (GinMaster)

• Discuss cases that makes analysis difficult

• Also applies to similar non exploit Android samples.
Andr/MstrKey-A- Overwriting?

- Trivial but Important in the case of this Exploit analysis
- Might end up analysing the wrong file

```java
inflating: assets/fashion.jspx
inflating: res/layout/main.xml
inflating: **AndroidManifest.xml**
replace **AndroidManifest.xml**? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

```java
inflating: res/drawable-hdpi/icon.png
inflating: res/drawable-ldpi/icon.png
inflating: res/drawable-mdpi/icon.png
inflating: **classes.dex**
replace **classes.dex**? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```
**Andr/MstrKey-A- Duplicate files!**

- Watch out if you want to use Apktool d <apk>

<table>
<thead>
<tr>
<th>File Size</th>
<th>Date/Time</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2320255</td>
<td>21/07/2013</td>
<td>assets/fashion.jxp</td>
</tr>
<tr>
<td>640</td>
<td>21/07/2013</td>
<td>res/layout/main.xml</td>
</tr>
<tr>
<td>6780</td>
<td>21/07/2013</td>
<td>AndroidManifest.xml</td>
</tr>
<tr>
<td>1404</td>
<td>21/07/2013</td>
<td>AndroidManifest.xml</td>
</tr>
<tr>
<td>1476</td>
<td>21/07/2013</td>
<td>resources.arsc</td>
</tr>
<tr>
<td>7542</td>
<td>21/07/2013</td>
<td>res/drawable-hdpi/icon.png</td>
</tr>
<tr>
<td>2760</td>
<td>21/07/2013</td>
<td>res/drawable-ldpi/icon.png</td>
</tr>
<tr>
<td>4278</td>
<td>21/07/2013</td>
<td>res/drawable-mdpi/icon.png</td>
</tr>
<tr>
<td>90080</td>
<td>21/07/2013</td>
<td>classes.dex</td>
</tr>
<tr>
<td>4908</td>
<td>21/07/2013</td>
<td>classes.dex</td>
</tr>
<tr>
<td>635</td>
<td>21/07/2013</td>
<td>META-INF/MANIFEST.MF</td>
</tr>
<tr>
<td>688</td>
<td>21/07/2013</td>
<td>META-INF/CERT.SF</td>
</tr>
<tr>
<td>1007</td>
<td>21/07/2013</td>
<td>META-INF/CERT.RSA</td>
</tr>
</tbody>
</table>
Andr/DroidD - Dex header

- IDA Pro 6.4adv default installation displayed corrupt message
Andr/DroidD - Dex header

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0 – 0x7</td>
<td>DEX_FILE_MAGIC</td>
</tr>
</tbody>
</table>

DEX_FILE_MAGIC = "dex\n<Version number>\0"

- 0x64 0x65 0x78 0x0a 0x30 0x33 0x36 0x00 = “dex.036”
- Version ‘036’ – Current version ( 4.x )
- Version ‘035’ – older api level 13 and earlier, most malware had this version number in our DB.

http://www.strazzere.com/blog/2013/02/loose-documentation-leads-to-easy-disassembler-breakages
Unfamiliar Opcodes

• Familiar technique employed in PC world

• Simple but still effective way

• Lot of research already done and now see them being used

• Packers, malware use to break tools and slow analysis
Unfamiliar Opcodes

- Dex2jar and baksmali v 1.4.1* failed to work

Caused by: java.lang.RuntimeException: **opcode format for 64 not found!**
  at com.googlecode.dex2jar.readerOpcodeFormat.get(OpcodeFormat.java:362)
  at ... 8 more

*Works in baksmali version 2.03*
Unfamiliar Opcodes

- **40** - unused\_40*
- **64** - Reads the char static field identified by the field\_id into vx.
- **6400 0200** - sget-byte v0, Test3.bs1:B // field@0002
  Reads byte field@0002 (entry #2 in the field id table) into v0

*src -http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html
Decompilation

- Decompilation is preferred method of analysis.
- Dex2jar followed by jdgui/jad is the most used method.
- Junk insertion, obfuscation, not maintained.

```java
// ERROR //
public final void uncaughtException(java.lang.Thread paramThread, java.lang.Throwable paramThrowable)
{
    // Byte code:
    // 0: new 80     java/lang/StringBuilder
    // 3: dup
    // 4: bipush 254
}

Caused by: java.lang.ArrayIndexOutOfBoundsException: 105
```
Outcome?

- Popular tools aren’t enough for complete analysis
- Tools update still catching up with research Implementation
- Support for tools maintenance and development by Android/Google
- IDE for malware analysis ??????
- Better to combine with dynamic Analysis
Questions...