Hiding Behind ART
Agenda

Introduction

ART Overview

User Mode Rootkits

Demo

Conclusion
Motivation

- Recent advancements in Android security
  - dm-verity
    - allows Android to verify the integrity of a partition at boot time
    - detect modifications in /system
    - protects devices from rootkits that adds or modifies binaries in the /system partition
    - not yet enabled by default
  - What can an attacker do despite of this?
  - Can we conduct rootkit operations without touching /system?
Introduction

Approach

- To answer these questions, we turned to ART

- Take advantage of ART’s mechanisms to modify framework and app code without touching /system
Agenda

Introduction
ART Overview
User Mode Rootkits
Demo
Conclusion
Background

- Introduced in Android KitKat 4.4 back in October, 2013

- Became the default runtime in Android Lollipop 5.0 in November 2014
Background

- Dalvik
  - Interpreted
  - Dexopt
  - Just-in-time (JIT) compilation

- ART
  - Ahead-of-time (AOT) compilation
  - Dalvik bytecode -> Native code
Background

- Advantages
  - Better performance
  - Better battery life

- Some very minor drawbacks
  - More storage space
  - Longer installation time
When?

- At first boot or system upgrade
  - Creates boot.oat and boot image
  - All installed apps will be compiled
  - May take a while

- Upon app installation/update
Dex2oat

- Dex2oat
  - Ex:
    
    ```
    /system/bin/dex2oat --zip-fd=6 --zip-location=/system/app/Chrome/Chrome.apk --oat-fd=7 --oat-location=/data/dalvik-cache/arm/system@app@Chrome@Chrome.apk@classes.dex --instruction-set=arm --instruction-set-features=default --runtime-arg -Xms64m --runtime-arg -Xmx512m --swap-fd=8
    ```

- Compiles bytecode in classes.dex into native code
- Resulting OAT file will be placed in /data/dalvik-cache/<target architecture>
- When app is run, the code generated in the resulting OAT file is executed instead of the bytecode in the DEX
Compilation

- Compiler backends:
  - Quick
  - Portable

- “--compile-backend” option for dex2oat

- Current default is Quick
Quick Backend

- Medium level IR (DEX bytecode)
- Low level IR
- Native code
- Some optimizations at each stage
Portable backend

- Uses LLVM bitcode as its LIR
- Optimizations using LLVM optimizer
- Code generation is done by LLVM backends
Boot.oat

- system@framework@boot.oat
- Contains libs and frameworks in boot class path
  – To be pre-loaded in all apps

```
```
ART Overview

**Boot.oat**

- /system/framework/core-libart.jar
- /system/framework/conscrypt.jar
- /system/framework/okhttp.jar
- /system/framework/core-junit.jar
- /system/framework/bouncycastle.jar
- /system/framework/ext.jar
- /system/framework/framework.jar
- /system/framework/framework.jar:classes2.dex
- /system/framework/telephony-common.jar
- /system/framework/voip-common.jar
- /system/framework/ims-common.jar
- /system/framework/mms-common.jar
- /system/framework/android.policy.jar
- /system/framework/apache-xml.jar
Boot image

- `system@framework@boot.art`
- Contains pre-initialized classes and objects from the framework
- Contains pointers to methods in boot.oat
- `boot.oat` and app oat contain pointers to methods in the boot image
- Loaded by zygote along with boot.oat
### Layout

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Permissions</th>
<th>Offset</th>
<th>Size</th>
<th>File Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>70070000-709e2000</td>
<td>rw-p</td>
<td></td>
<td></td>
<td>/data/dalvik-cache/arm/system@<a href="mailto:framework@boot.art">framework@boot.art</a></td>
</tr>
<tr>
<td>709e2000-7246f000</td>
<td>r--p</td>
<td></td>
<td></td>
<td>/data/dalvik-cache/arm/system@<a href="mailto:framework@boot.oat">framework@boot.oat</a></td>
</tr>
<tr>
<td>7246f000-739a5000</td>
<td>r-xp</td>
<td></td>
<td></td>
<td>/data/dalvik-cache/arm/system@<a href="mailto:framework@boot.oat">framework@boot.oat</a></td>
</tr>
<tr>
<td>739a5000-739a6000</td>
<td>rw-p</td>
<td></td>
<td></td>
<td>/data/dalvik-cache/arm/system@<a href="mailto:framework@boot.oat">framework@boot.oat</a></td>
</tr>
</tbody>
</table>
## ART Image Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>ubyte[4]</td>
<td>Image version</td>
</tr>
<tr>
<td>image_begin</td>
<td>uint32</td>
<td>Base address of the image</td>
</tr>
<tr>
<td>image_size</td>
<td>uint32</td>
<td>The size of the image</td>
</tr>
<tr>
<td>image_bitmap_offset</td>
<td>uint32</td>
<td>Offset to a bitmap</td>
</tr>
<tr>
<td>image_bitmap_size</td>
<td>uint32</td>
<td>Size of the image bitmap</td>
</tr>
<tr>
<td>oat_checksum</td>
<td>uint32</td>
<td>Checksum of the linked boot.oat file</td>
</tr>
<tr>
<td>oat_file_begin</td>
<td>uint32</td>
<td>Address of the linked boot.oat file</td>
</tr>
<tr>
<td>oat_data_begin</td>
<td>uint32</td>
<td>Address of the linked boot.oat file’s oadata</td>
</tr>
<tr>
<td>oat_data_end</td>
<td>uint32</td>
<td>End address of the linked boot.oat file’s oadata</td>
</tr>
<tr>
<td>oat_file_end</td>
<td>uint32</td>
<td>End address of the linked boot.oat file</td>
</tr>
<tr>
<td>patch_delta</td>
<td>int32</td>
<td>Image relocated address delta</td>
</tr>
<tr>
<td>image_roots</td>
<td>uint32</td>
<td>Address of an array of objects</td>
</tr>
<tr>
<td>compile_pic</td>
<td>uint32</td>
<td>Indicates if image was compiled with position-independent-code enabled</td>
</tr>
</tbody>
</table>
## OAT File

- ELF dynamic object
- .oat/.dex file extension

```plaintext
<table>
<thead>
<tr>
<th>struct dynamic_symbol_table</th>
<th>[U] &lt;Undefined&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ struct Elf32_Sym symtab[0]</td>
<td></td>
</tr>
<tr>
<td>◀ struct sym_name32_t sym_name</td>
<td>oatdata</td>
</tr>
<tr>
<td>Elf32_Addr sym_value</td>
<td>0x00001000</td>
</tr>
<tr>
<td>Elf32_Xword sym_size</td>
<td>892928</td>
</tr>
<tr>
<td>▶ struct sym_info_t sym_info</td>
<td>STB_GLOBAL</td>
</tr>
<tr>
<td>unsigned char sym_other</td>
<td>0</td>
</tr>
<tr>
<td>Elf32_Half sym_shndx</td>
<td>4</td>
</tr>
<tr>
<td>▶ char sym_data[892928]</td>
<td></td>
</tr>
<tr>
<td>▶ struct Elf32_Sym symtab[1]</td>
<td></td>
</tr>
<tr>
<td>◀ struct sym_name32_t sym_name</td>
<td>oatexec</td>
</tr>
<tr>
<td>Elf32_Addr sym_value</td>
<td>0x000DB000</td>
</tr>
<tr>
<td>Elf32_Xword sym_size</td>
<td>605104</td>
</tr>
<tr>
<td>▶ struct sym_info_t sym_info</td>
<td>STB_GLOBAL</td>
</tr>
<tr>
<td>unsigned char sym_other</td>
<td>0</td>
</tr>
<tr>
<td>Elf32_Half sym_shndx</td>
<td>5</td>
</tr>
<tr>
<td>▶ char sym_data[605104]</td>
<td></td>
</tr>
<tr>
<td>▶ struct Elf32_Sym symtab[2]</td>
<td></td>
</tr>
<tr>
<td>◀ struct sym_name32_t sym_name</td>
<td>oatexec</td>
</tr>
<tr>
<td>Elf32_Addr sym_value</td>
<td>0x0016EBAC</td>
</tr>
<tr>
<td>Elf32_Xword sym_size</td>
<td>4</td>
</tr>
<tr>
<td>▶ struct sym_info_t sym_info</td>
<td>STB_GLOBAL</td>
</tr>
<tr>
<td>unsigned char sym_other</td>
<td>0</td>
</tr>
<tr>
<td>Elf32_Half sym_shndx</td>
<td>5</td>
</tr>
</tbody>
</table>
```
OAT File

- Dynamic symbol tables pointing to OAT data and code
  - `oatdata`
  - `oatexec`
  - `oatlastword`
### OAT File

- **oatdata** -> headers, DEX files
- **oatexec** -> compiled code
- **oatlastword** -> end marker

```plaintext
`struct dynamic_symbol_table {
  `struct Elf32_Sym symtab[0] {
    Elf32_Addr sym_value 0x00001000
    Elf32_Xword sym_size 892928
    struct sym_info_t sym_info
    unsigned char sym_other 0
    Elf32_Half sym_shndx 4
    char sym_data[892928]
  }
  `struct Elf32_Sym symtab[1] {
    Elf32_Addr sym_value 0x0000DB00
    Elf32_Xword sym_size 605104
    struct sym_info_t sym_info
    unsigned char sym_other 0
    Elf32_Half sym_shndx 5
    char sym_data[605104]
  }
  `struct Elf32_Sym symtab[2] {
    Elf32_Addr sym_value 0x0016EBAC
    Elf32_Xword sym_size 4
    struct sym_info_t sym_info
    unsigned char sym_other 0
    Elf32_Half sym_shndx 5
    char sym_data[4]
  }
  `struct Elf32_Sym symtab[3] {
    Elf32_Addr sym_value
    Elf32_Xword sym_size
    struct sym_info_t sym_info
    unsigned char sym_other
    Elf32_Half sym_shndx
    char sym_data[4]
  }
}
`
OAT File

- OAT Header
- OAT DEX File Header [0]
- OAT DEX File Header [n]
- DEX File [0]
- DEX File [n]
- OAT Class [0]
- OAT Class [0]

---

oatdata (.rodata)

oatexec (.text)

---

OAT Code
## OAT Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adler32_checksum</td>
<td>uint32</td>
<td>Adler-32 checksum of the OAT header</td>
</tr>
<tr>
<td>instruction_set</td>
<td>uint32</td>
<td>Instruction set architecture</td>
</tr>
<tr>
<td>instruction_set_features</td>
<td>uint32</td>
<td>Bitmask of supported features per architecture</td>
</tr>
<tr>
<td>dex_file_count</td>
<td>uint32</td>
<td>Number of DEX files in the OAT</td>
</tr>
<tr>
<td>executable_offset</td>
<td>uint32</td>
<td>Offset of executable code section from start of oatdata</td>
</tr>
<tr>
<td>interpreter_to_interpreter_bridge_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to interpreter_to_interpreter_bridge stub</td>
</tr>
<tr>
<td>interpreter_to_compiled_code_bridge_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to interpreter_to_compiled_code_bridge stub</td>
</tr>
<tr>
<td>jni_dlsym_lookup_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to jni_dlsym_lookup stub</td>
</tr>
<tr>
<td>portable_imt_conflict_trampoline_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to portable_imt_conflict_trampoline_stub</td>
</tr>
<tr>
<td>portable_resolution_trampoline_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to portable_resolution_trampoline_stub</td>
</tr>
<tr>
<td>portable_to_interpreter_bridge_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to portable_to_interpreter_bridge stub</td>
</tr>
<tr>
<td>quick_generic_jni_trampoline_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to quick_generic_jni_trampoline_stub</td>
</tr>
<tr>
<td>quick_imt_conflict_trampoline_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to quick_imt_conflict_trampoline_stub</td>
</tr>
<tr>
<td>quick_resolution_trampoline_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to quick_resolution_trampoline_stub</td>
</tr>
<tr>
<td>quick_to_interpreter_bridge_offset</td>
<td>uint32</td>
<td>Offset from oatdata start to quick_to_interpreter_bridge_stub</td>
</tr>
<tr>
<td>image_patch_delta</td>
<td>int32</td>
<td>The image relocated address delta</td>
</tr>
<tr>
<td>image_file_location_oat_checksum</td>
<td>uint32</td>
<td>Adler-32 checksum of boot.oat’s header</td>
</tr>
<tr>
<td>image_file_location_oat_data_begin</td>
<td>uint32</td>
<td>The virtual address of boot.oat’s odata section</td>
</tr>
<tr>
<td>key_value_store_size</td>
<td>uint32</td>
<td>The length of key_value_store</td>
</tr>
<tr>
<td>key_value_store</td>
<td>ubyte[key_value_store_size]</td>
<td>A dictionary containing information such as the command line used to generate this oat file, the host arch, etc.</td>
</tr>
</tbody>
</table>
## OAT Header

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kNone</td>
<td>0</td>
<td>Unspecified</td>
</tr>
<tr>
<td>kArm</td>
<td>1</td>
<td>ARM</td>
</tr>
<tr>
<td>kArm64</td>
<td>2</td>
<td>ARM 64-bit</td>
</tr>
<tr>
<td>kThumb2</td>
<td>3</td>
<td>Thumb-2</td>
</tr>
<tr>
<td>kX86</td>
<td>4</td>
<td>X86</td>
</tr>
<tr>
<td>X86_64</td>
<td>5</td>
<td>X64</td>
</tr>
<tr>
<td>kMips</td>
<td>6</td>
<td>MIPS</td>
</tr>
<tr>
<td>kMips64</td>
<td>7</td>
<td>MIPS 64-bit</td>
</tr>
</tbody>
</table>
OAT Dex File Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dex_file_location_size</td>
<td>uint32</td>
<td>Length of the original input DEX path</td>
</tr>
<tr>
<td>dex_file_location_data</td>
<td>ubyte[dex_file_location_size]</td>
<td>Original path of input DEX file</td>
</tr>
<tr>
<td>dex_file_location_checksum</td>
<td>uint32</td>
<td>CRC32 checksum of classes.dex</td>
</tr>
<tr>
<td>dex_file_pointer</td>
<td>uint32</td>
<td>Offset of embedded input DEX from start of oatdata</td>
</tr>
<tr>
<td>classes_offsets</td>
<td>uint32[DEX.header.class defs size]</td>
<td>List of offsets to OATClassHeaders</td>
</tr>
</tbody>
</table>

- Original DEX file is embedded at offset `dex_file_pointer`
- Size of classes_offsets corresponds to the class_defs_size field of the DEX file’s header
OAT Class Header

- Type indicates how much of the methods were compiled (https://source.android.com/devices/tech/dalvik/configure.html)

- If type == kOatSomeCompiled, there will be a bitmap_size and bitmap field

- Each bit in the bitmap represents a method of this class

- A set bit bit means, this method was compiled

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>uint16</td>
<td>State of class during compilation</td>
</tr>
<tr>
<td>type</td>
<td>uint16</td>
<td>Type of class</td>
</tr>
<tr>
<td>bitmap_size</td>
<td>uint32</td>
<td>Size of compiled methods bitmap (present only when type = 1)</td>
</tr>
<tr>
<td>bitmap</td>
<td>ubyte[bitmap_size]</td>
<td>Compiled methods bitmap (present only when type = 1)</td>
</tr>
<tr>
<td>methods_offsets</td>
<td>uint32[variable]</td>
<td>List of offsets to the native code for each compiled method</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kOatClassAllCompiled</td>
<td>0</td>
<td>All methods in the class are compiled.</td>
</tr>
<tr>
<td>kOatClassSomeCompiled</td>
<td>1</td>
<td>Some methods are compiled.</td>
</tr>
<tr>
<td>kOatClassNoneCompiled</td>
<td>2</td>
<td>No methods were compiled.</td>
</tr>
</tbody>
</table>
OAT Class Header

• Each method_offset points to the generated native method code.

• Take note that for kThumb2 architecture, code_offset has the least significant bit set.
  - Ex: For method_offset 0x00143061, the actual start of the native code is at offset 0x00143060.
OAT Quick Method Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapping_table_offset</td>
<td>uint32</td>
<td>Offset from the start of the mapping table</td>
</tr>
<tr>
<td>vmap_table_offset</td>
<td>uint32</td>
<td>Offset form the start of the vmap table</td>
</tr>
<tr>
<td>gc_map_offset</td>
<td>uint32</td>
<td>Offset to the GC map</td>
</tr>
<tr>
<td>QuickMethodFrameInfo.frame_size_in_bytes</td>
<td>uint32</td>
<td>Frame size for this method when executed</td>
</tr>
<tr>
<td>QuickMethodFrameInfo.core_spill_mask</td>
<td>uint32</td>
<td>Bitmap of spilled machine registers</td>
</tr>
<tr>
<td>QuickMethodFrameInfo.fp_spill_mask</td>
<td>uint32</td>
<td>Bitmap of spilled floating point machine registers</td>
</tr>
<tr>
<td>code_size</td>
<td>uint32</td>
<td>The size of the generated native code</td>
</tr>
</tbody>
</table>

- Generated for Quick backend compiled code
- Mapping between registers and ip in native code and Dalvik bytecode
Agenda

Introduction

ART Overview

User Mode Rootkits

Demo

Conclusion
Approach

- Use dex2oat to generate OAT files from modified framework or app and replace the originals
- Replace framework code
  - Generate new boot.art and boot.oat and replace the system generated one
- Replace application code
  - Generate new OAT and replace the installed app’s OAT
- Requires a root shell
Advantages

- No low level code required
  - Code modifications are done in Java
  - Less problems encountered than dealing with low level kernel stuff

- Less affected by variations in architecture and OS version
  - Same approach works regardless of the arch and OS

- We don’t have to deal with code signing
  - Apps are already installed and verified
Advantages

- Our code runs under the context of the app running it
- Same uid and app permissions
- Example: Settings app
  - system uid
  - Permissions:

  - android.permission.REBOOT
  - android.permission.MANAGE_DEVICE_ADMIN
  - android.permission.MANAGE_USERS
  - android.permission.WRITE_SECURE_SETTINGS
  - android.permission.MOUNT_UNMOUNT_FILESYSTEMS
  - android.permission.ACCESS_NOTIFICATIONS
  - android.permission.CLEAR_APP_USER_DATA
Persistence

- Our code persists for as long as the OAT file is not replaced

- Our goal is not to maintain root access
  – no writes to /system, remember?
  – We do have the option to re-acquire root access using a system-to-root exploit (when running as system)
Replacing framework code

- Replace framework code with our own
- Use dex2oat to generate a new boot.art and boot.oat that includes our modified JAR
- Replace original boot.oat with our own boot.oat
Replacing framework code

- What we want to do
  - Hide running processes
  - Hide files
  - Hide installed apps
  - and more...
## Replacing framework code

### – Target methods

<table>
<thead>
<tr>
<th>What to hide</th>
<th>Class</th>
<th>Method</th>
<th>Source</th>
<th>JAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running processes</td>
<td>ActivityManager</td>
<td>getRunningAppProcesses</td>
<td>/frameworks/base/core/java/android/app/ActivityManager.java</td>
<td>framework.jar</td>
</tr>
<tr>
<td>Installed apps</td>
<td>ApplicationPackageManager</td>
<td>getInstalledApplications</td>
<td>/frameworks/base/core/java/android/app/ApplicationPackageManager.java</td>
<td>framework.jar</td>
</tr>
<tr>
<td>Files</td>
<td>File</td>
<td>filenamesToFiles</td>
<td>/libcore/luni/src/main/java/java/io/File.java</td>
<td>core-libart.jar</td>
</tr>
</tbody>
</table>
Replacing framework code

- **Example: Hide running processes**
  
  - `ActivityManager.getRunningAppProcesses()`
  
  - Source code can be found in “/frameworks/base/core/java/android/app/ActivityManager.java”
  
  - Build results in /system/framework.jar
Replacing framework code

```java
public List<RunningAppProcessInfo> getRunningAppProcesses() {
    try {
        return ActivityManagerNative.getDefault().getRunningAppProcesses();
    } catch (RemoteException e) {
        return null;
    }
}
```

- Returns a list of RunningAppProcessInfo
- We need to modify the list
Replacing framework code

```java
public List<RunningAppProcessInfo> getRunningAppProcesses() {
    try {
        List<RunningAppProcessInfo> procList = ActivityManagerNative.getDefault().getRunningAppProcesses();

        for (Iterator<RunningAppProcessInfo> iter = procList.listIterator(); iter.hasNext();)
            RunningAppProcessInfo p = iter.next();
        if (p.processName.equals("com.polsab.badapp")) {
            iter.remove();
        }

        return procList;
    } catch (RemoteException e) {
        return null;
    }
}
```
Replacing framework code

- Build the modified code
  - We only use this JAR to get the smali code for the modified method

- Use apktool to decode the resulting JAR

- Locate the generated smali for the method
Replacing framework code

- **Step 1: Modify target method**
  - Pull the original JAR from the /system partition.
  - Use apktool to decode the JAR and generate smali code.
  - Modify the target method(s).
  - Rebuild the JAR using apktool.
Replacing framework code

- Step 2: Prepare the JAR

  - Rename the JAR such that the resulting path after you have pushed it to the device is the same length with the path of the original JAR in the /system partition.

    “/system/framework/framework.jar”
    “/data/local/tmp/11framework.jar”

  - Makes relocating offsets unnecessary.
Replacing framework code

- Step 3: Get checksum of the original classes.dex
  
  - Get the CRC32 of classes.dex in the original JAR.
  
  - We will patch this to our OAT later.
Replacing framework code

- Step 4: Prepare to run dex2oat
  
  - Delete the original boot.oat.
  
  - Push our modified JAR into the device
  
  - Retrieve the command line used to generate the original boot.oat.
    
    • Get this from key_value_store in the OAT header
Replacing framework code

- **Step 5: Generate our boot.oat**

  Replace all references to our target JAR with the path of our modified JAR:

  ```
  ```

- **Run dex2oat**
Replacing framework code

- Step 6: Patch boot.oat DEX path and checksum
  - Once boot.oat is generated, patch the `dex_file_location_data` with the original JAR’s path.
  - Patch the `dex_file_location_checksum`, which is right after the path, with the original classes.dex’s checksum we calculated earlier.
Replacing framework code

- Step 7: Restart Zygote

  – For the changes to take effect, we have to restart Zygote or restart the device.

    ```
    stop zygote
    start zygote
    ```

  – Installed apps will be recompiled
Replacing app code

- Modify specific apps instead of a system framework JAR.
- Affects only a single app, so less intrusive than replacing boot.oat

Downsides:
- It only affects apps you specifically target
- Apps are updated more frequently
  - System apps, not so much
Replacing app code

- Example: Settings.apk
  - Shows running processes and installed apps
  - Original APK is in “/system/priv-app/Settings/Settings.apk”
  - Source code in AOSP’s package/apps/Settings
Replacing app code

- To hide our app from the running processes list
  - Look for calls to `ActivityManager.getRunningAppProcesses()`
  - Modify the returned `RunningAppProcessInfo` list.
Replacing app code

- packages/apps/Settings/src/com/android/settings/applications/RunningState.java

```java
List<ActivityManager.RunningAppProcessInfo> processes = am.getRunningAppProcesses();

for (Iterator<ActivityManager.RunningAppProcessInfo> iter = processes.listIterator(); iter.hasNext();)
    ActivityManager.RunningAppProcessInfo p = iter.next();

if (p.processName.equals("com.polsab.badapp")) {
    iter.remove();
}
```
Replacing app code

- To hide our app from installed apps list
  - Look for calls to
    PackageManager.getInstalledApplications()
  - Modify the returned ApplicationInfo list.
Replacing app code

- packages/apps/Settings/src/com/android/settings/applications/ApplicationState.java

```java
mApplications = mPm.getInstalledApplications(mRetrieveFlags);
if (mApplications == null) {
    mApplications = new ArrayList<ApplicationInfo>();
}

for (Iterator<ApplicationInfo> iter = mApplications.listIterator(); iter.hasNext(); ) {
    ApplicationInfo a = iter.next();
    if (a.processName.equals("com.polsab.badapp")) {
        iter.remove();
    }
}
```
Replacing app code

- Step 1: Modify target method
  - Pull the original APK from its install location.
  - Use apktool to decode the APK and generate smali code.
  - Modify the target method(s).
  - Rebuild the APK.
Replacing app code

- **Step 2: Prepare the APK**

  - Rename the APK such that the resulting path after you have pushed it to the device is the same length with the path of the original APK.

    
    - 
    - `/system/priv-app/Settings/Settings.apk`
    - `/data/local/tmp/1111111111Settings.apk`

  - Makes relocating offsets unnecessary.
Replacing app code

- Step 3: Get checksum of the original classes.dex
  - Get the CRC32 of classes.dex in the original APK.
  - We will patch this to our OAT later.
Replacing app code

- Step 4: Prepare to run dex2oat
  - Delete the original OAT file.
  - Push our modified APK to the device
Replacing app code

Step 5: Generate our OAT

- Run the dex2oat command with the following parameters:
  - --dex-file = <our modified APK’s path>
  - --oat-file = <original OAT file’s path>

- Example:

  ```
  dex2oat -dex-file=/data/local/temp/111111111Settings.apk -oat-file=/data/dalvik-cache/arm/system@priv-app@Settings@Settings.apk@classes.dex
  ```
Replacing app code

- Step 6: Patch OAT file’s DEX path and checksum
  
  - Once the OAT file is generated, patch the `dex_file_location_data` with the original APK’s path.
  
  - Patch the `dex_file_location_checksum`, which is right after the path, with the original classes.dex’s checksum we calculated earlier.
Replacing app code

- Step 7: Restart the app

  - Stop the app process if it is running.
    - Ex:
      ```
      am force-stop com.android.settings
      ```

  - The changes will take effect the next time the app is run.
Limitations

- We can’t hide from lower level or non-framework code

- SELinux policies may stop us
  - Not a problem if you can setenforce 0

- Your code is bound by the affected app’s permissions
Agenda

Introduction

ART Overview

User Mode Rootkits

Demo

Conclusion
Agenda

Introduction
ART Overview
User Mode Rootkits
Demo
Conclusion
Conclusion

- User mode rootkits are possible through ART – You can use these techniques for RE as well

- We can still achieve persistence on the device

- ART is ripe for more security research
Questions?
Thanks for listening!

Paul Sabanal