# Introduction to Dynamic Dalvik Instrumentation

SummerCon 2013

Collin Mulliner, June 2013, New York City

crm[at]ccs.neu.edu

**NEU SECLAB**

# $ finger collin@mulliner.org

- 'postdoc' Security Researcher
    - $HOME = Northeastern University, Boston, MA, USA
    - `cat .project`
            specialized in *mobile handset security*


- Current work
    - Android security
    - Android security


- Past work
    - Bluetooth security
    - A lot on SMS and MMS security
    - Mobile web usage and privacy
    - Some early work on NFC phone security

# Introduction

- Android Application Security
  - Find vulnerabilities (audit)
  - Analyze malware
  - RE ... what is this application doing
  - ATTACK stuff

- What does this thing do? How does this thing work?
  - Disassemble → look at smali code
  - Run in emulator/sandbox → look at traces / network
  - (Static) instrumentation → look at app while it runs

# Introduction

- Android Application Security
  - Find vulnerabilities (audit)
  - Analyze malware
  - RE ... what is this application doing
  - ATTACK stuff

- What does this thing do? How does this thing work?
  - Disassemble → look at smali code
  - Run in emulator/sandbox → look at traces / network
  - (Static) instrumentation → look at app while it runs

- **This talk is about Dynamic Instrumentation**
  - **Instrumentation at the Dalvik level**
  (but not bytecode!)

**NEU SECLAB**

# Static Instrumentation on Android

- Unpack APK
    - Convert manifest back to plain text, ...

- Disassemble DEX classes
    - Get smali code

- Instrument smali code
    - Modify smali code, add own code

- Repackage application
    - Compile code, Sign, etc...

- Install and run
    - Hope it works... (bug in patch, self integrity check, ...)

**NEU SECLAB**

# Dynamic Instrumentation

- <u>Change/modify application code at runtime</u>
  - Allows to add and remove code/hooks on-the-fly
  - Technique has been around for many years

- Instrument library calls: quick overview what happens
  - No disassembly needed

- Still need to disassemble for target specific stuff
  - Find the interesting stuff to instrument

# Dynamic Instrumentation on Android

- No: unpacking, compile, repacking
  - Saves us time

- APK not modified
  - Defeat 'simple' integrity checks

- But Android Apps are written in Java and run in a VM...

**NEU SECLAB**

# Android
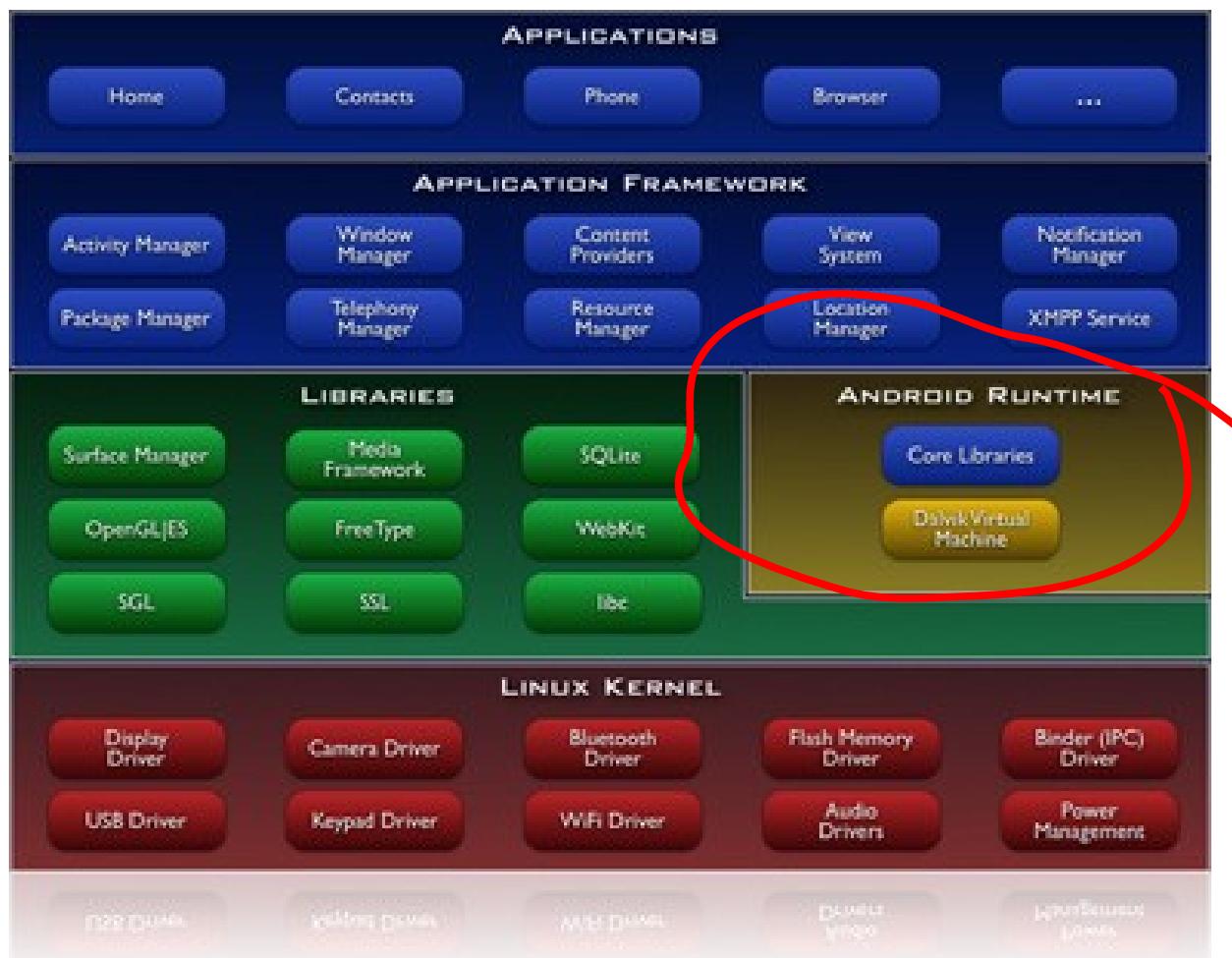
# Android Runtime

Android Process

- Dalvik Virtual Machine (DVM)
  Core Libraries (java.x.y)
  - Executes: Framework and Applications

  Dalvik Classes

- Application
  - Process for "MainActivity"
  - Additional process(s) for "Service"

  libdvm

- Framework works in the same way!
  - zygote
  - system_server
  - ...

  libc, libz, libjpeg,...
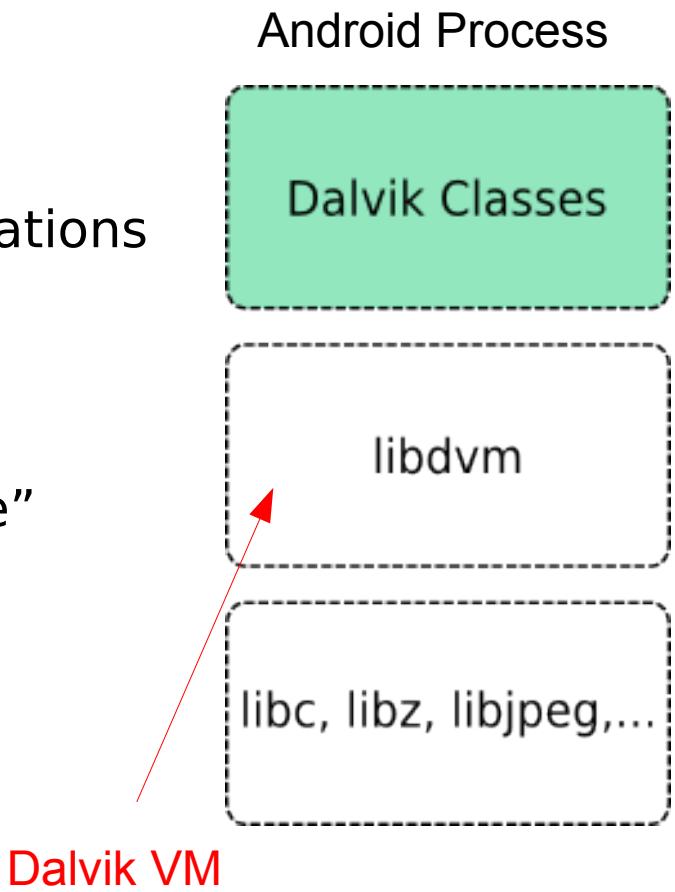
Dalvik VM

# Dalvik Instrumentation – The Basic Idea

- Convert Dalvik method to native (JNI) method
    - We get control of the execution

- Call original Dalvik method from native method
    - This creates an in-line hook of the Dalvik method

- Implement instrumentation code using JNI
    - Access to everything
    **(private, protected doesn't exist in the land of C)**

# Java Native Interface (JNI) super quick intro

- C API to interact between the Java and C/native world
  - You can write any type of java code using JNI ;-)

- JNI function, signature: `result name(JNIEnv *env, …)`
  - Callable from the Java world

- JNI is essential for our instrumentation!
  - Need to know this in order to do instrumentation!
    (but not to understand the talk!)

```
FindClass()        // obtain class reference
NewObject()        // create a new class object
GetMethodId()      // get method
CallObjectMethod() // call a method
...
```

**NEU SECLAB**

# Dalvik Instrumentation – Overview

Android Process

- Inject 'shared object' (.so) into running process
  - Provides the native code
  - My talk: *Dynamic Binary Instrumentation on Android (SummerCon 2012)*

Dalvik Classes

Do stuff to DVM

libdvm

- Native code 'talks to the DVM'
  - Resolve symbols from DVM
  - Call DVM functions to:
    - Lookup classes and methods
    - Hook method
    - Call original method

libc, libz, libjpeg,...

injected library
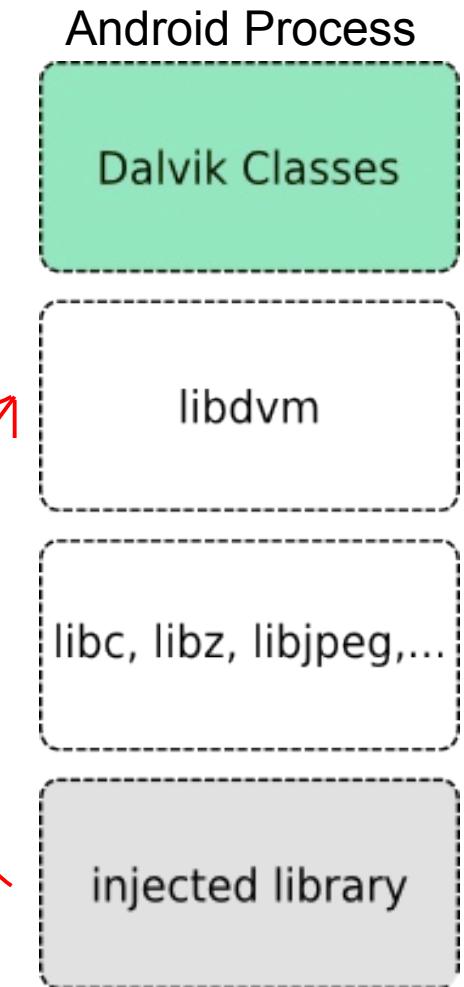
# Hooking a Dalvik Method 1/3

- Find loaded class

- Find method by name and signature

- Change method parameters

- Convert to JNI method

```
cls = dvmFindLoadedClass("Ljava/lang/String;");
met = dvmFindVirtualMethodHierByDescriptor(cls, "compareTo",
                            "(Ljava/lang/String;)I");
```

*if direct method use: dvmFindDirectMethodByDescriptor()

# Hooking a Dalvik Method 2/3

- Method parameters (interesting for our task)

```
insSize            // size of input parameters
outSize            // size of output
registersSize      // size of method bytecode
insns              // bytecode
JniArgInfo         // argument parsing info (JNI)
access flags       // public, protected, private, native :-)
```

- *insSize* and *registersSize* are set to a specific value (next slides)
- *outSize* = 0
- *insns* is saved for calling original function (next slides)
- *JniArgInfo* = 0x80000000 (→ parse method arguments)
- *access flags* = access flags | 0x0100 (make method native)

**NEU SECLAB**

# Hooking a Dalvik Method 3/3

- Convert to JNI method

```
int dalvik_func_hook(JNIEnv *env, jobject this, jobject str)
{
 ...
}


dvmUseJNIBridge(met, dalvik_func_hook);
```

- Every call to `java.lang.String.compareTo(String)` is now handled by `dalvik_func_hook()`

# Method Parameter Manipulation : the details

- The DVM needs to know how *big* the method arguments are
  - insSize
  - We also set registersSize == insSize

- Argument size calculation
  - Every argument adds one (1) to the input size
  - J (a double) adds two (2)
  - For methods of object classes (non static classes) add one (1) for the instance (this)

```
java.lang.String.compareTo("Ljava/lang/String;)I
insSize == 2
```

# Calling the Original Method

- Lookup class + method (or used saved values from hooking)

- Revert method parameters (or used saved values)

- Call method → inspect result → hook method again

```
int dalvik_hook_func(JNIEnv *env, jobject this, jobject str)
{
    jvalue args[1];
    args[0].l = str;
    int res = (*env)->CallIntMethodA(env, this, meth, args);
    return res;
}
```

# LibDalvikHook 1/2

- Easy to use Dalvik hooking library
  - Provides: hooking, unhooking, calling original method

```
struct dalvik_hook_t h;     // hook data, remembers stuff for you

// setup the hook
dalvik_hook_setup(
    &h,                          // hook data
    "Ljava/lang/String;",       // class name
    "compareTo",                 // method name
    "(Ljava/lang/String;)I",    // method signature
    2, // insSize (need to calculate that in your head! LOL)
    hook_func_compareto         // hook function
);

// place hook
dalvik_hook(&libdhook, &h);
```

**NEU SECLAB**

# LibDalvikHook 2/2

- Calling the original method

```
int hook_func(JNIEnv *env, …)
{
  dalvik_prepare(
    &libdhook,        // library context
    &h,               // hook data
    env               // JNI environment
  );
  // use JNI API to call method
  args[0].l = x;
  CallXXMethod(env, obj, h.mid, args); // h.mid → method

  dalvik_postcall(&libdhook, &h);
}
```

- Unhook by simply only calling **dalvik_prepare()**

# Injecting the Instrumentation Library 1/2

- hijack tool from my talk about native Android instrumentation
  - SummerCon 2012

- Steps:
  - Push library and DEX file to /data/local/tmp
  - Enable DEX loading (`chmod 777 /data/dalvik-cache/`)
  - hijack -p PID -l /data/local/tmp/lib.so

- Injects the library into running process
  - Works on any process, including system apps + services e.g. zygote, system_server, ... :-)

# Injecting the Instrumentation Library 2/2

- We want to inject into processes before they are execute
    – All Dalvik processes are forked from zygote

- hijack zygote and inject when it specializes
    – Need to know the main class of target application
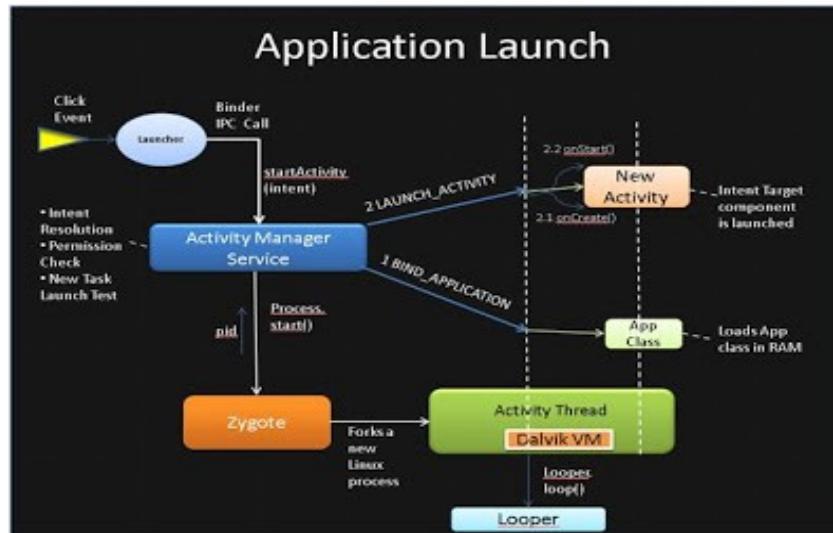


```
hijack -p zygotePID -l lib.so -s org.mulliner.collin.work
```

Collin Mulliner – SummerCon 2013  "Introduction to Dynamic Dalvik Instrumentation"

# Hijack's newest Features

- Inject into zygote  **-z**

- Inject into new DVM process by class name (combine wit -z)
    **-s full.class.name**

- Disable calling mprotect() before injecting, old Android versions
    **-m**

- Debug level switch
    **-D <level>**

# Instrumentation Code Flow (v1)

Method in App (Java)

Hook (JNI function)

Original function (Java)

proxy
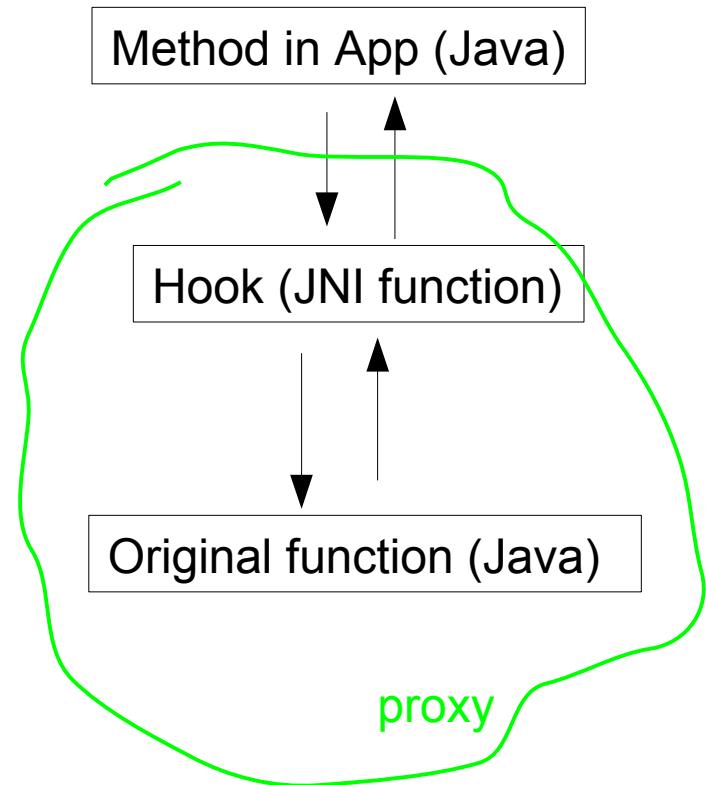
# Monitor / Reverse Applications

- How does the application work?
  - Maybe App is obfuscated, strings are "encrypted"

- Instrument interesting methods to see what App does
  - String operations
  - Reflection
  - ...

```
String   java.lang.StringBuffer.toString()
int      java.lang.String.compareTo(..)
int      java.lang.String.compareToIgnoreCase(..)
String   java.lang.StringBuilder.toString()


Method   java.lang.Class.getMethod(..)
```

# Attack "Stuff"

- Disable Signature Verification
  - Used for all kinds of things...
  - Patch to always "return true;"
    (used it to attack various things)

```
boolean java.security.Signature.verify(byte[]) { … }
```

# Loading Additional Classes

- Sophisticated "instrumentation"
  - way easier done in Java then in C-JNI
  - You really want to be able to write stuff in Java
    if you want to interact with the Android framework

- Loading classes is supported by LibDalvikHook
  - dexstuff_loaddex()
  - dexstuff_defineclass()

# Loading Classes 1/3

- Load DEX file into DVM
- Define classes, tell DVM what classes to load from DEX file
    - Get class loader...

```
args[0].l = "PATH/classes.dex"; // must be a string object
cookie = dvm_dalvik_system_DexFile[0](args, &pResult);

// get class loader
Method *m = dvmGetCurrentJNIMethod();
// define class
u4 args[] = {
  "org.mulliner.collin.work", // class name (string object)
  m->clazz->classLoader,      // class loader
  cookie                      // use DEX file loaded above
};
dvm_dalvik_system_DexFile[3](args, &pResult);
```

**NEU SECLAB**

# Loading Classes 2/3



benign process     libddi injected     ddi classes loaded

calls JNI method

load class and call method

loaded Dalvik code

call DVM to hook stuff

**NEU SECLAB**
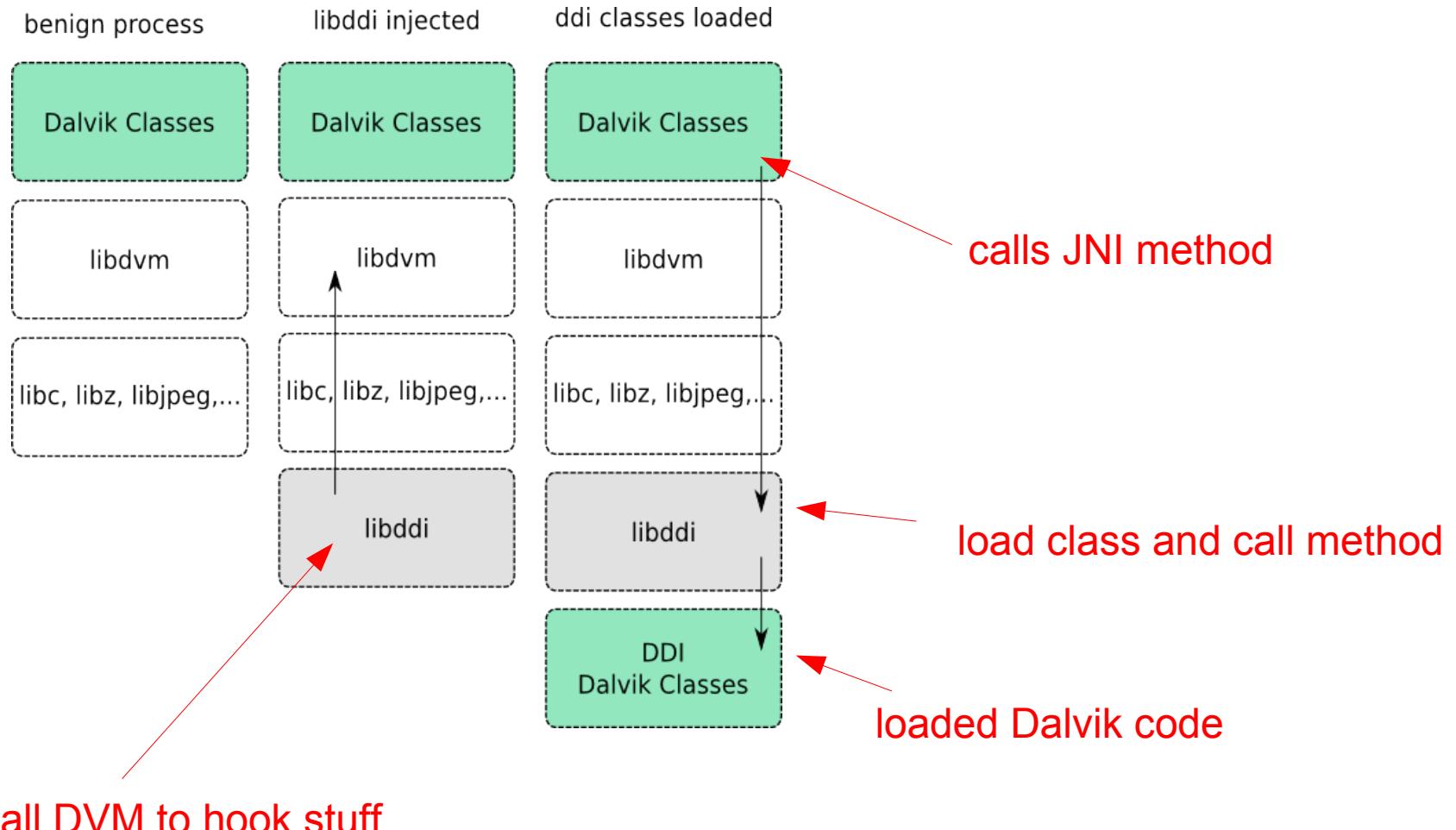
# Loading Classes 3/3

- The loaded classes can be used like any other class
    - Using C-JNI or Java code


- Each class has to be defined (incl. all inner classes), yes really!
    - e.g. `org.mulliner.collin.`**`work$really`**


- Dalvik cache at: `/data/dalvik-cache`
    - Needs to be made world writable
    Required for class loader to write **odex** file

    - **odex** file needs to be deleted on class update
    `rm /data/dalvik-cache/data@local@tmp@classes.dex`

# Instrumentation Code Flow (v2)

Method in App (Java)

Hook (JNI function)

Original function (Java)

proxy

Load

Instrumentation Class(s) (Java)

Method in Instrument (Java)

Sophisticated instrumentation framework interaction
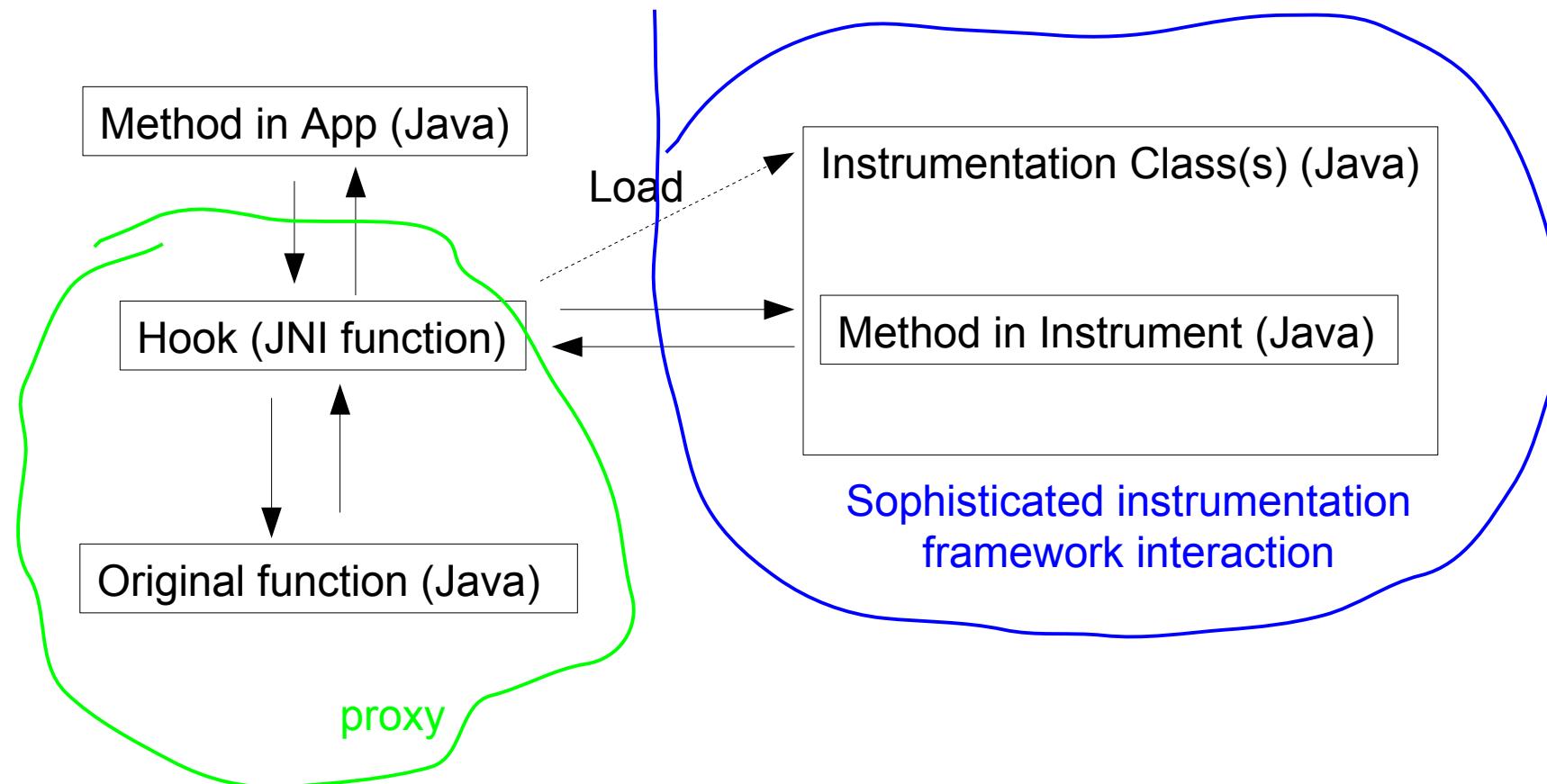
# Interacting with the Target Application

- Our (java) code runs inside the target process, yay!
    - But how do we interact with it?

- Access target's objects (class instances)
    - Scrape them from method parameters

    ```
    int somemethod(Intent x, CustomClass y)
    ```

- Access the Application Context (android.content.Context)
    - Interact with the Android framework: send Intents, … (next slides)

# Field Scraping 1/2

- Access fields (class variables)
  - Manipulate and/or extract data

- Steps
  - Acquire class object (e.g. thru method hook)
  - Know the field name and type
    (source or disassembly of target class)
  - Access field (JNI GetXField)

```
jobject some_method(JNIEnv *env, jobject obj, …)
{
  cls = FindClass(env, "org/mulliner/collin/work");
  fid = GetFieldID(env, cls, "fieldname",
            "Landroid/content/Context;");
  jobject = GetObjectField(env, obj, fid);
```

**NEU SECLAB**

# Field Scraping 2/2 (for java nerds)

- Inner vs. outer Class
  - Sometimes you will have access to wired stuff
    but not the stuff you are looking for
  - e.g access to some inner class (ending with `$Name`)
    you want the outer class or some member of it

- Java generates synthetic member variables for you
  - Inner class has access to the outer class via `this$0`

```
org.mulliner.collin.work & org.mulliner.collin.work$harder

Access only to object of type $harder

FindClass(env, "org/mulliner/collin/work$harder);
GetFieldID(env, cls, "this$0", "Lorg/mulliner/collin/work");
```

**NEU SECLAB**

# Access to Application Context

- Scrape fields of type: Service, Application, …
  - Say hi to your disassembler :)

- Use the ActivityThread
  - Usable from any UI thread

```
Class<?> activityThreadClass =
            Class.forName("android.App.ActivityThread");

Method method =
      activityThreadClass.getMethod("currentApplication");

Application app =
            (Application) method.invoke(null, (Object[])null);
```

# Rapid Prototyping of Framework Modifications

- Defense against SMS OTP stealing Trojans [1]
  - Change local SMS routing based on SMS content

- For the prototype we needed to change code in the framework

```
com/android/internal/telephony/SMSDispatcher.java
protected void dispatchPdus(byte[] pdus) { … }
```

- Instead of recompiling Android just replace the method
  → save a lot of time
  → test on many different devices without custom compile

[1] *SMS-based One-Time Passwords: Attacks and Defense (short paper)* Collin Mulliner, Ravishankar Borgaonkar, Patrick Stewin, Jean-Pierre Seifert
To appear In the Proceedings of the 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment
(DIMVA 2013) Berlin, Germany, July 2013

**NEU SECLAB**

# Using DVM internal functions, for profit

- Dump list of loaded classes in current VM
  - Useful to find out which system process runs a specific framework service

```
dvmDumpAllClasses(level);
// level  0 = only class names 1 = class details
```

- Dump details of specific class
  - All methods (incl. signature), fields, etc...

```
cls = dvmFindLoadedClass("Lorg/mulliner/collin/work");
dvmDumpClass(cls, 1);
```

# DvmDumpClass output for java.lang.String

```
I/dalvikvm(  410): ----- class 'Ljava/lang/String;' cl=0x0 ser=0x50000016 -----
I/dalvikvm(  410):   objectSize=24 (8 from super)
I/dalvikvm(  410):   access=0x0003.0011
I/dalvikvm(  410):   super='Ljava/lang/Object;' (cl=0x0)
I/dalvikvm(  410):   interfaces (3):
I/dalvikvm(  410):     0: Ljava/io/Serializable; (cl=0x0)
I/dalvikvm(  410):     1: Ljava/lang/Comparable; (cl=0x0)
I/dalvikvm(  410):     2: Ljava/lang/CharSequence; (cl=0x0)
I/dalvikvm(  410):   vtable (62 entries, 11 in super):
I/dalvikvm(  410):     17: 0x56afd4e8                compareTo (Ljava/lang/String;)I
I/dalvikvm(  410):     18: 0x56afd520  compareToIgnoreCase (Ljava/lang/String;)I
I/dalvikvm(  410):     19: 0x56afd558                   concat (Ljava/lang/String;)...
I/dalvikvm(  410):     20: 0x56afd590                 contains (Ljava/lang/CharSequ...
I/dalvikvm(  410):     21: 0x56afd5c8            contentEquals (Ljava/lang/CharSequ...
. . . .
I/dalvikvm(  410):   static fields (4 entries):
I/dalvikvm(  410):     0:                     ASCII [C
I/dalvikvm(  410):     1: CASE_INSENSITIVE_ORDER Ljava/util/Comparator;
I/dalvikvm(  410):     2:     REPLACEMENT_CHAR C
I/dalvikvm(  410):     3:     serialVersionUID J
I/dalvikvm(  410):   instance fields (4 entries):
I/dalvikvm(  410):     0:                     value [C
I/dalvikvm(  410):     1:                  hashCode I
I/dalvikvm(  410):     2:                    offset I
```

**NEU SECLAB**

# Modifying Stuff Globally

- **zygote** is base VM for all processes
  - Code injected into zygote propagates to all newly created processes

- **system_server** handles like everything
  - monitor and/or cross process Intents



Application Launch
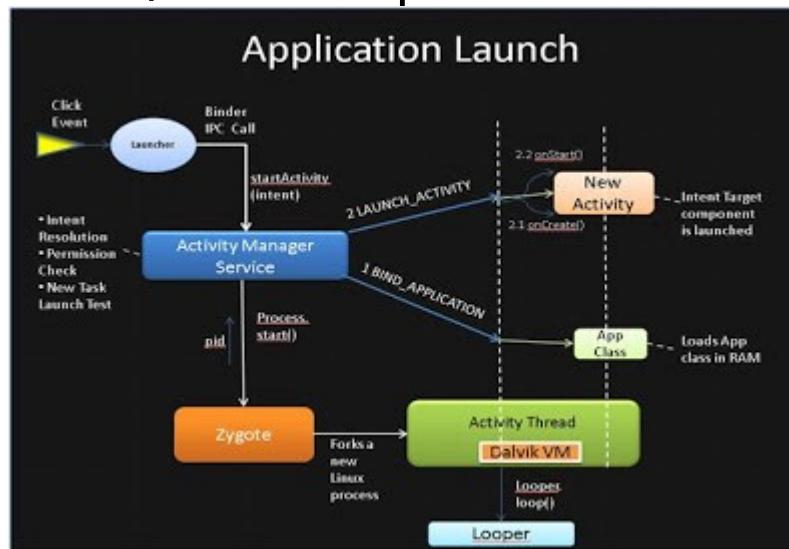
# Getting Serious!

- We can...
  - inject native + Dalvik code into any Android process
  - hook Dalvik methods in Apps, the Framework, and Java core libraries
  - Interact with the Apps and the Android framework

- We did...
  - spy on behavior of Apps API calls
  - changed SMS handling in the Android framework

- **Lets attack real stuff and make some $$$$**

# Conclusions

- Dynamic Instrumentation via the Android Runtime allows
  - Modification of Apps and the Framework in memory
  - Doesn't break APK signatures
  - Portable across devices
  - Super stable (not a hack)
  - But can only replace whole functions
    - no bytecode modification

- Possible to stir up Android AppSec quite a bit
  - Obfuscation and use of reflection is kinda useless

- We have various ongoing projects based on this
  - Students doing interesting stuff

**NEU SECLAB**

**EOF**

# Thank you!

twitter: @collinrm
crm[at]ccs.neu.edu
http://mulliner.org/android

# The Dalvik VM - libdvm

- We interrogate the DVM using dlsym()
  - We just need a small number of symbols

```
// hooking
dvmFindLoadedClass
dvmFindVirtualMethodHierByDescriptor
dvmFindDirectMethodByDescriptor
dvmUseJNIBridge
// class loading
dvm_dalvik_system_DexFile
dvmStringFromCStr
dvmGetSystemClassLoader
dvmGetCurrentJNIMethod
// debugging :)
dvmDumpAllClasses
dvmDumpClass
```

**NEU SECLAB**