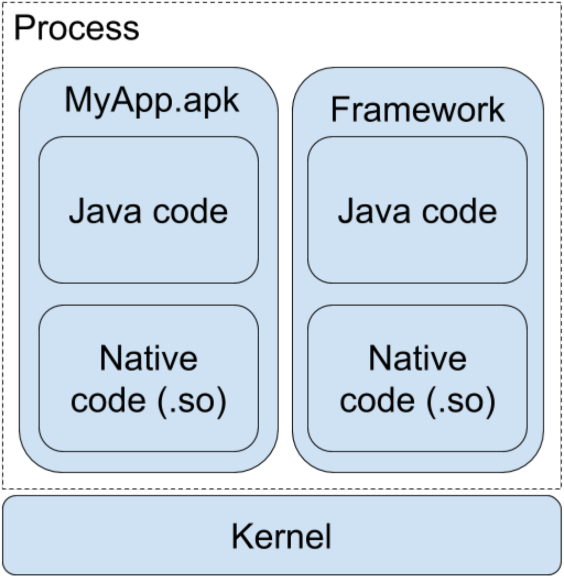
Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy

Vitor Monte Afonso [1], Antonio Bianchi [2], Yanick Fratantonio [2], Adam Doupé [3], Mario Polino [4], Paulo Lício de Geus [1], Christopher Kruegel [2], and Giovanni Vigna [2]

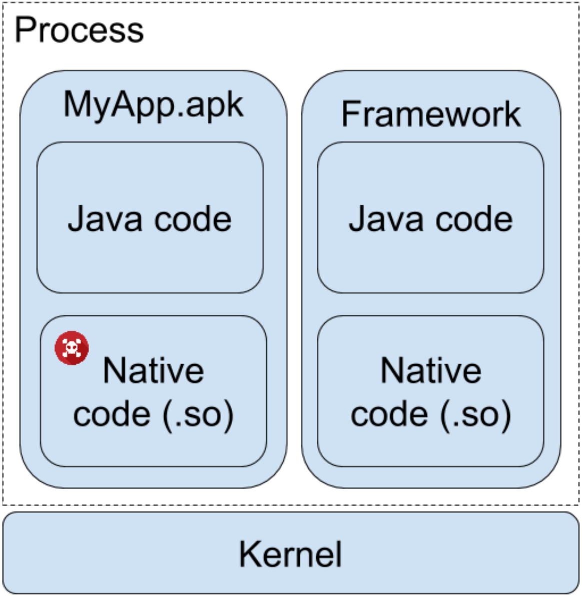
[1] University of Campinas
[2] UC Santa Barbara
[3] Arizona State University
[4] Politecnico di Milano

NDSS 2016

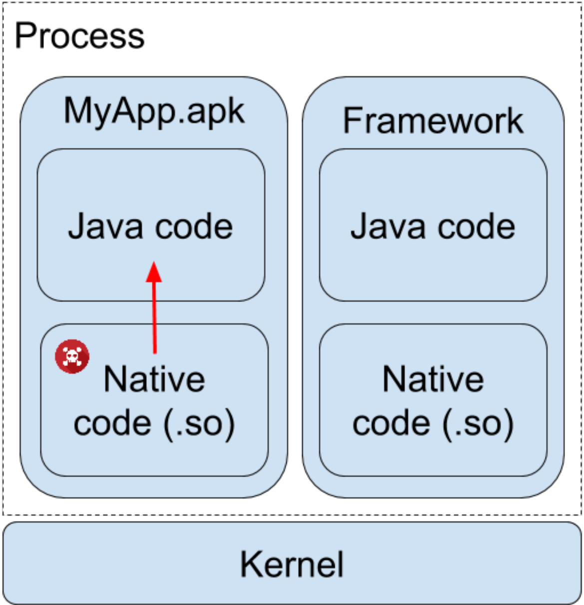
Introduction



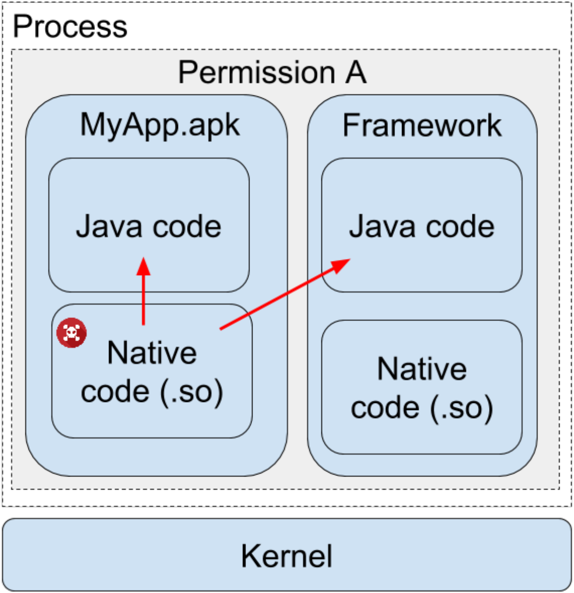
Introduction



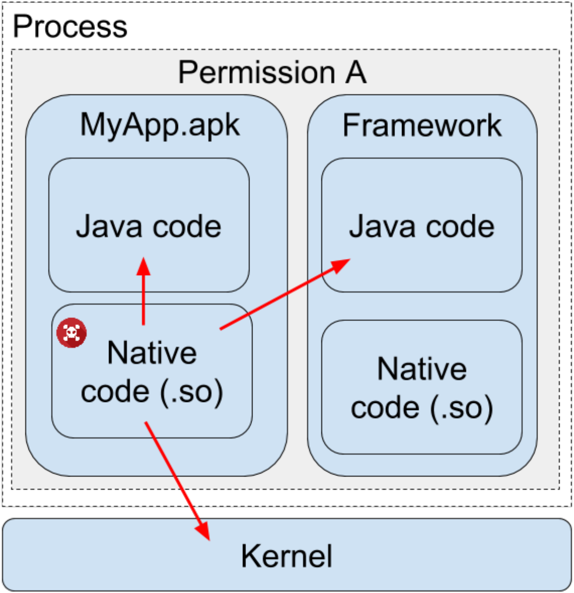
Introduction



Introduction

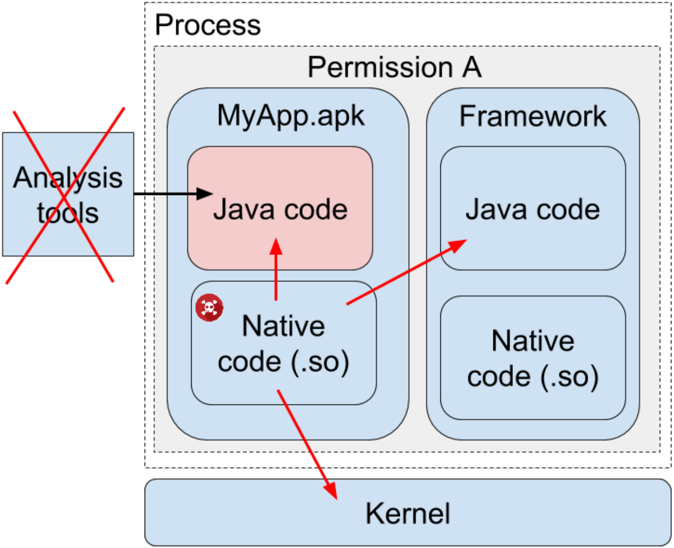


Introduction

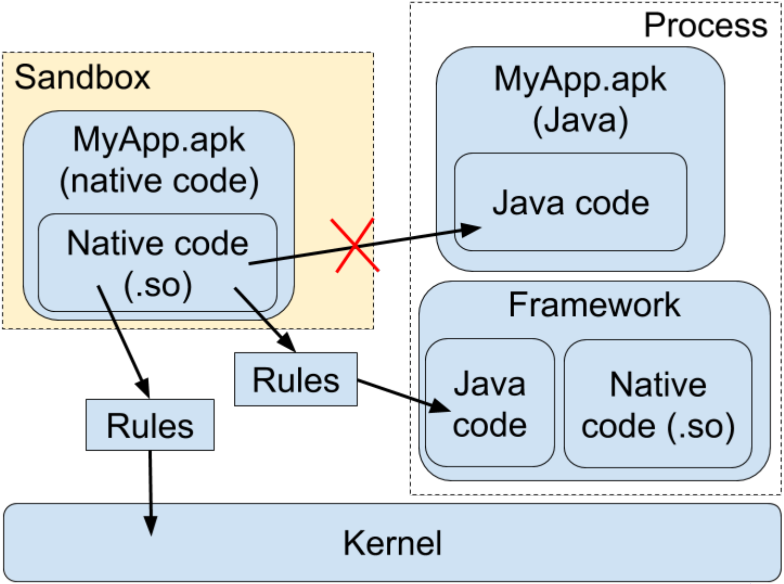


Introduction

- Most analysis tools miss these attacks



Introduction - Sandboxing



Introduction

Motivation

- Lack of data regarding native code usage
- No research on how to generate a general, practical and useful policy to enforce

Introduction

Motivation

- Lack of data regarding native code usage
- No research on how to generate a general, practical and useful policy to enforce

Large-scale analysis

- How many apps actually use native code?
- What is the behavior of native code?
- What permissions do native code use?
- How does native code interact with the app and the framework?
- Which shared libraries are used in native code?

Background

Native code

- Executable file
  - Exec methods (Runtime.exec or ProcessBuilder.start)
- Shared library (.so)
  - Load methods (e.g., System.loadLibrary)
  - Native methods
  - Native activity

Applications Used

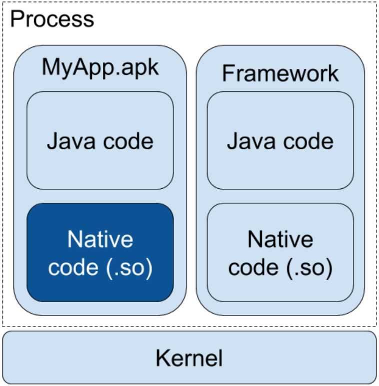
Dataset

- 1,208,476 distinct free apps
- Crawled from Google Play - May 2012 and August 2014

Static prefiltering

- Filtered apps that have the potential to use native code
  - Native method: Java method with “native” modifier
  - Native activity: declared in manifest or class that extends NativeActivity
  - Call to Exec or Load methods
  - ELF file inside APK
- 37.0% (446,562) have the potential to use native code

Dynamic Analysis



Information to track

- System calls of native code
- Interactions of native code with other components

Dynamic Analysis

Our system

- App's system calls traced with `strace`
- Instrumented libraries
  - Flag third-party libraries (based on file path)
  - Record all transitions between Java and app's native code
- Post-processing - separate behavior of app's native code

Research Question

How many apps actually use native code?

Dynamic Analysis

- 33.6% (149,949) of dynamically analyzed apps executed native code
- 12.4% of all apps in our dataset - other work identified around 5%
- It's only a lower bound: it could be more

Apps	Type
72,768	Native method
19,164	Native activity
132,843	Load library
27,701	Call executable file (27,599 standard, 148 custom and 46 both)
149,949	At least one of the above

Native Code Not Reached

Small experiment

- Manual analysis
- 20 random apps
- Static analysis
  - 40% (8) deadcode - native code unreachable from Java code
- Other apps were very complex
  - Dynamically analyzed those and interacted manually
  - Still did not reach native code

Why deadcode

- Third-party libraries - include a lot of code but only part of it is used

Research Question

What is the behavior of native code?

Native Code Behavior - Overview

Common actions in shared libraries

- 94.2% (125,192) of apps that used custom shared libs only performed subset of common actions
- Such as memory management system calls, calling JNI functions, writing log messages and creating directories

Other actions in shared libs and custom executable files

- Most common are: `ioctl` calls, writing file in app's directory, operations on sockets

Standard executable files

- Most common are: read system information, write file in app's dir or sdcard, read logcat

Research Question

What permissions do native code use?

Top 5 Permissions Used in Native Code

Apps	Permission	Description
1,818	INTERNET	Open network socket or call method `java.net.URL.openConnection`
1,211	WRITE_EXTERNAL_STORAGE	Write files to the sdcard
1,211	READ_EXTERNAL_STORAGE	Read files from the sdcard
132	READ_PHONE_STATE	Call methods `getSubscriberId`, `getDeviceSoftwareVersion`, `getSimSerialNumber` or `getDeviceId` from class `android.telephony.TelephonyManager` or Binder transaction to call `com.android.internal.telephony.IPhoneSubInfo.getDeviceId`
79	ACCESS_NETWORK_STATE	Call method `android.net.ConnectivityManager.getNetworkInfo`

Research Question

How does native code interact with the app and the framework?

JNI Calls

- How native code interact with the app and the framework

Most common groups of JNI calls used

Apps	Description
94,543	Get class or method identifier and class reference
71,470	Get or destroy JavaVM, and Get JNIEnv
53,219	Manipulation of String objects
...	...
35,231	Call Java method (in app or framework)

Most common groups of methods from the Android framework called

Apps	Description
7,423	Get path to the Android package associated with the context of the caller
6,896	Get class name
5,499	Manipulate data structures
4,082	Methods related to cryptography

Research Question

Which shared libraries are used in native code?

Most Used Shared Libraries

Most used standard libraries

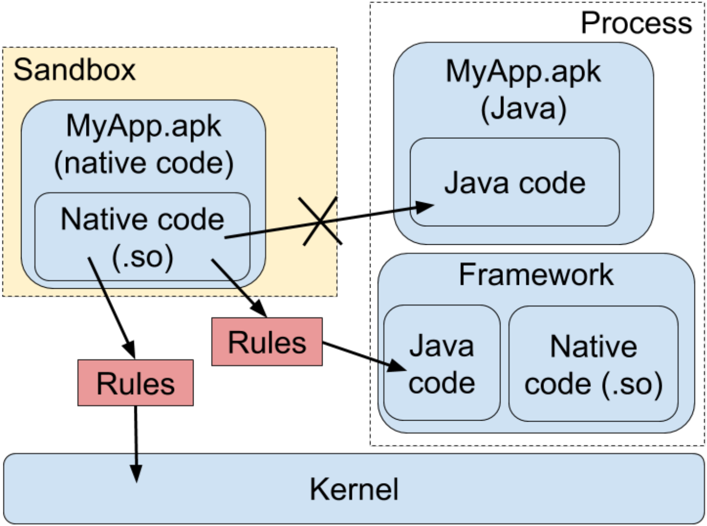
Apps	**Name**	**Description**
24,942	libjnigraphics.so	Manipulate Java bitmap objects
2,646	libOpenSLES.so	Audio input and output
2,645	libwilhelm.so	Multimedia output and audio input
349	libpixelflinger.so	Graphics rendering
347	libGLES_android.so	Graphics rendering

Most used custom libraries

Apps	**Name**	**Description**
19,158	libopenal.so	Rendering audio
17,343	libCore.so	Used by Adobe AIR
16,450	libmain.so	Common name
13,556	libstlport_shared.so	C++ standard libraries
11,486	libcorona.so	Part of the Corona SDK, a development platform for mobile apps

Sandboxing

- Now we can create the rules



Security Policy

Goal

- Reduce attack surface available for native code
- Generate security policy from data obtained

Trade-off

- Why not allowing everything?
- Overlap between benign and malicious behavior
- Tunable threshold: we selected 99%

Security Policy

Modes of operation

- Reporting or enforcing
- Not implemented

Process - system call policy

- Normalize arguments of system calls (e.g., file paths are replaced by “USER-PATH” or “SYS-PATH”)
- Iterate over syscalls
- Select the one used by most apps
- Repeat until allow certain percentage of apps to run

Root Exploits

Effects of policy with 99% threshold on root exploits

Name / CVE	Description	Blocked
Exploid (CVE-2009-1185)	Needs a `NETLINK` socket with `NETLINK_KOBJECT_UEVENT` protocol	Yes
GingerBreak (CVE-2011-1823)	Needs a `NETLINK` socket with `NETLINK_KOBJECT_UEVENT` protocol	Yes
CVE-2013-2094	Uses `perf_event_open` system call	Yes
Vold/ASEC	Creates symbolic link to a system directory	Yes
CVE-2013-6124	Creates symbolic links to system files	Yes
CVE-2011-1350	`ioctl` call used violates our rules	Yes
CVE-2011-1352	`ioctl` call used violates our rules	Yes
CVE-2012-4220	`ioctl` call used violates our rules	Yes
CVE-2012-4221	`ioctl` call used violates our rules	Yes
CVE-2012-4222	`ioctl` call used violates our rules	Yes
RATC (CVE-2010-EASY)	Relies on invoking many times the `fork` syscall	No
Zimperlinch	Relies on invoking many times the `fork` syscall	No
CVE-2011-1149	It relies on the `mprotect` syscall	No

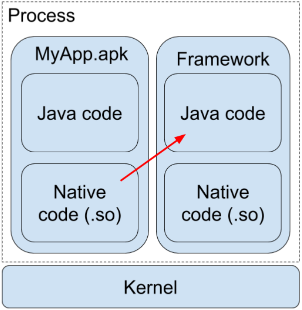
Root Exploits

Effects of policy with 99% threshold on root exploits

Name / CVE	Description	Blocked
Exploid (CVE-2009-1185)	Needs a `NETLINK` socket with `NETLINK_KOBJECT_UEVENT` protocol	Yes
GingerBreak (CVE-2011-1823)	Needs a `NETLINK` socket with `NETLINK_KOBJECT_UEVENT` protocol	Yes
CVE-2013-2094	Uses `perf_event_open` system call	Yes
Vold/ASEC	Creates symbolic link to a system directory	Yes
CVE-2013-6124	Creates symbolic links to system files	Yes
CVE-2011-1350	`ioctl` call used violates our rules	Yes
CVE-2011-1352	`ioctl` call used violates our rules	Yes
CVE-2012-4220	`ioctl` call used violates our rules	Yes
CVE-2012-4221	`ioctl` call used violates our rules	Yes
CVE-2012-4222	`ioctl` call used violates our rules	Yes
RATC (CVE-2010-EASY)	Relies on invoking many times the `fork` syscall	No
Zimperlinch	Relies on invoking many times the `fork` syscall	No
CVE-2011-1149	It relies on the `mprotect` syscall	No

- Collateral damage: 1,483 apps would be blocked

Java Method Security Policy



Java methods policy

- Performed same process to generate policy
- 99% threshold: 1,414 apps would be blocked
- Example of dangerous method that would be blocked if called from native code: `android.telephony.SmsManager.sendTextMessage`

Limitations

Dynamic analysis limitations

- Not all native code is executed
- In the real world apps might execute more than we observed in our experiments
- If our policy is adopted, it might block more apps

Possible improvements

- Use a more sophisticated tool to interact with the apps
- Track behavior in real devices

Summary

Advantage of large-scale experiments

- Since we analyzed a great amount of apps, we believe we observed most relevant behaviors

Security policies

- Based on behavior of many apps - first step to create usable policies

End

Questions ?

Vitor Monte Afonso - vitor@lasca.ic.unicamp.br