riscure

Introduction to Reverse Engineering Android applications

$ whoami

riscure

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

k3rckhoffs institute

riscure

Challenge your security

$ ps | grep –i hacking

riscure

- Hacking (@work)
  - Embedded hardware (STBs, PayTV chipsets, …)
  - Mobile apps (HCE/Mobile payment, DRM, …)
  - Obfuscation and white-box crypto solutions
  - Trusted Execution Environments

- Hacking (@home)
  - CTF’s (mainly reversing, exploiting and crypto)
  - Other random research

Reverse Engineering?

"Reverse engineering is the process of discovering the technological principles of a device, object, or system through analysis of its structure, function, and operation."

Source: Wikipedia.com

Why Reverse (Code) Engineering?

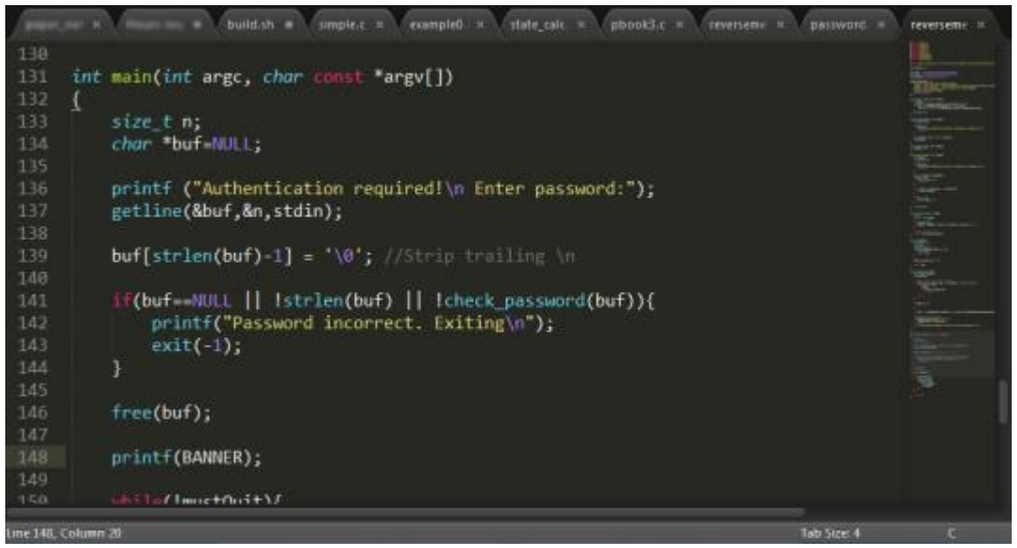
- System understanding allows identifying security weaknesses

- Requirement in black-box scenarios
  - This is how an attacker works anyway

- Vulnerabilities are introduced during compilation
  - Example: double-check removed by compiler

Compilation process

Source files



Compilation

Object files

Linking

Executable file

Reverse Engineering process

riscure

Executable file

Disassembly

Assembly representation

Analysis

Human readable *representation*

Binary code analysis

riscure

Static analysis:

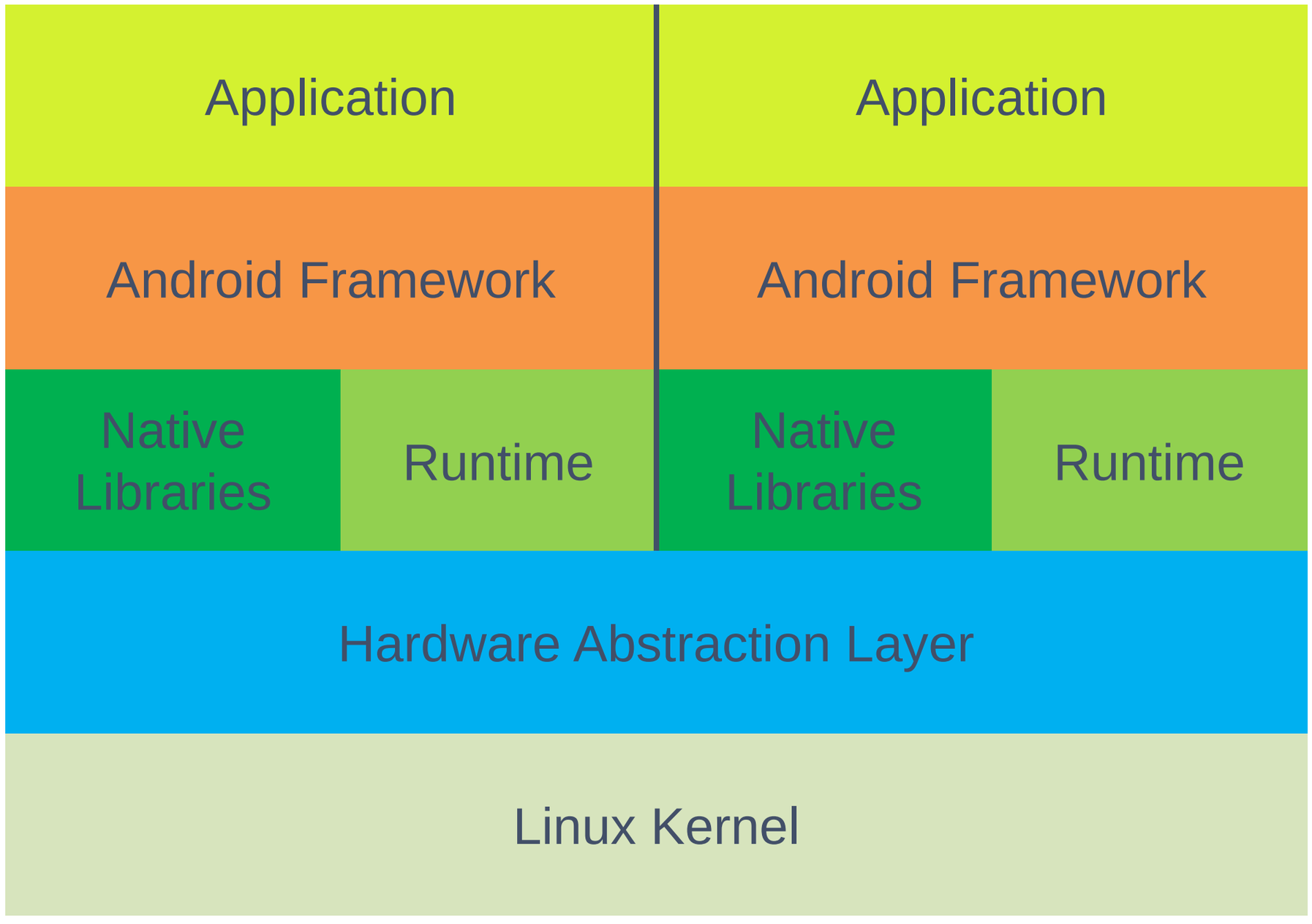
Reconstruct software behavior by analyzing code and data structures, without running or emulating the code itself.

Dynamic analysis:

Reconstruct software behavior by monitoring running code, interactions with external entities (files, network, etc.) and state of internal data structures.

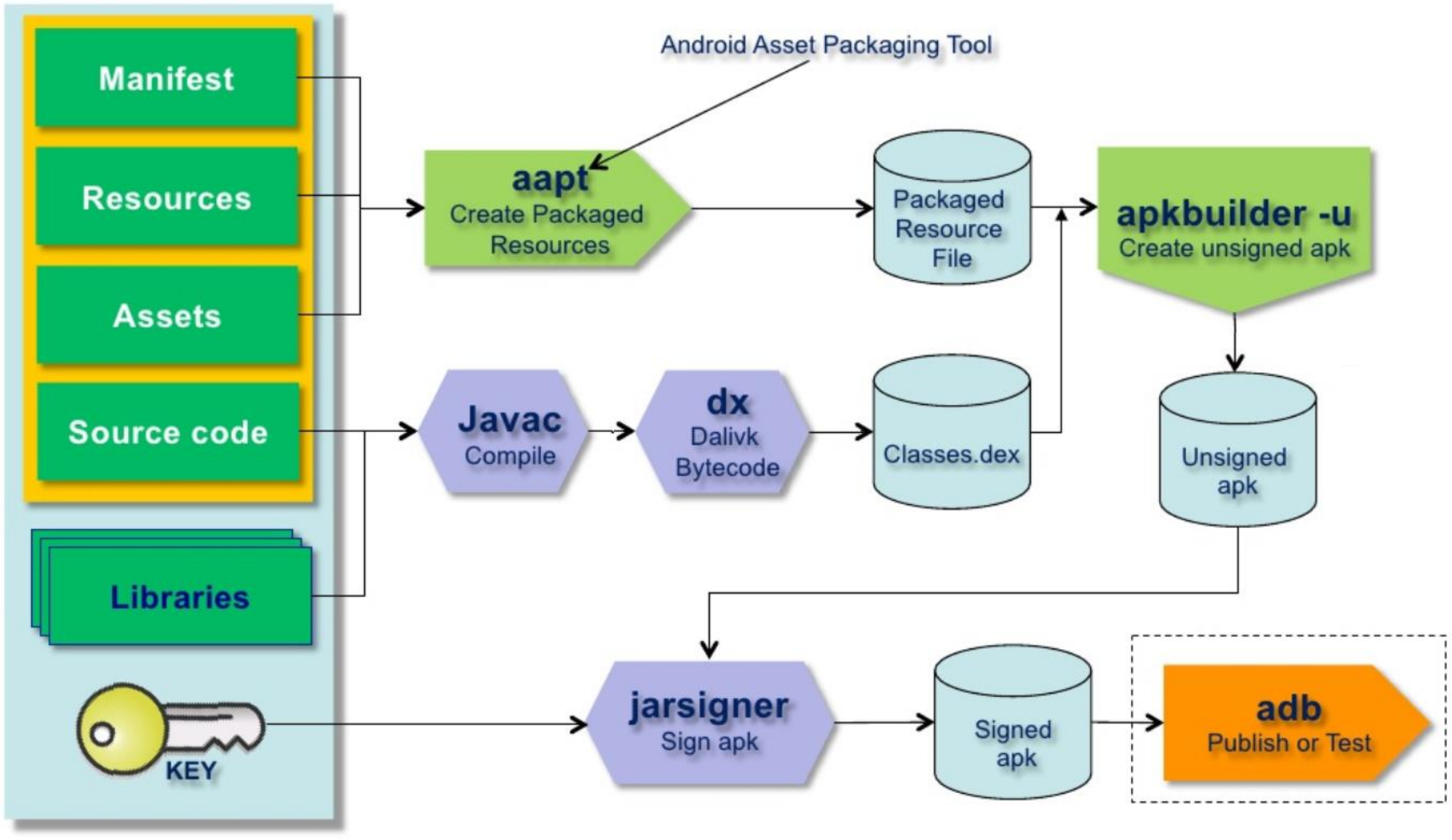
Android Introduction

Android architecture



Application development

riscure



Framework

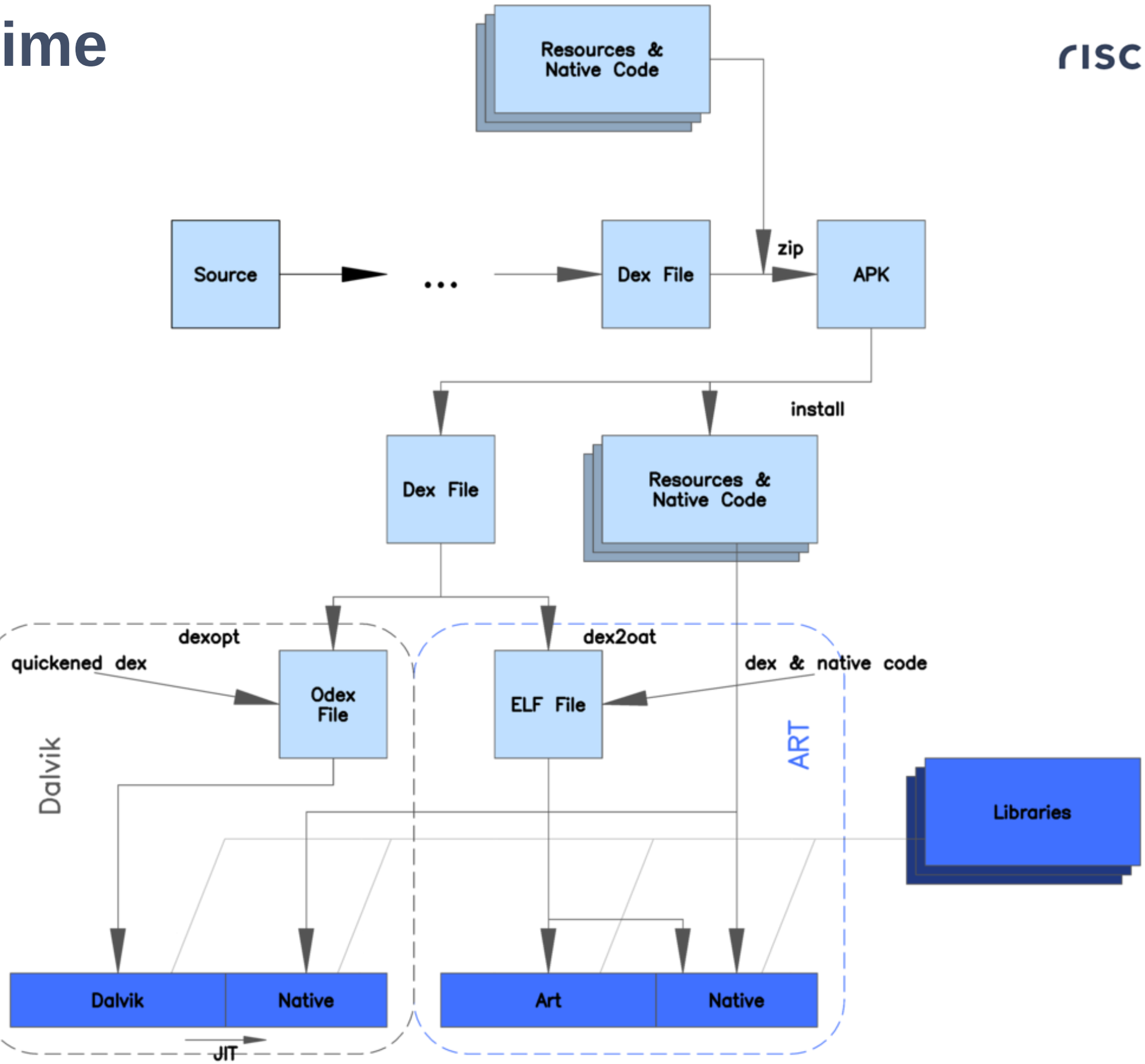
- Provides base functionality & resources for all apps
  - Includes java.* packages
- Can be customized by device vendor
  - Might be required to properly unpack app
- Apps “link” against a certain (minimum) level
- Framework not limited to a single apk

Runtime

- Virtual machine & sandbox
- Dalvik (original VM, removed in Android 5.0)
  - Starts with byte code interpreter
  - Trace based JIT compiler (Android 2.2+)
  - Device specific optimization with dexopt (dex → odex)
- ART (Android 4.4+)
  - Install time AOT compilation
  - ELF executable with native code

Runtime

riscure



Zygote

- Parent process of all (Dalvik) applications
- Starts at boot as one of the first processes
  - Loads the runtime & libraries
- Waits for a request to start a new application
  - Forks itself
    - Parent waits for the next request
    - Child loads application in preloaded VM
- Unmodified memory pages shared (copy on write)

Native Development Kit

- Dalvik code not as fast as native code
- Regular expressed wish to reuse existing libraries
- Solution: NDK
  - Introduced with Android 1.5
  - Allows code to be written in {arm, mips, x86} assembly, C, C++, …
  - Small set of prebuilt libraries included
    - Including a non-standard libc! (bionic)
    - 5 different C++ runtimes
  - JNI to call native code from Dalvik and vice versa

Java Native Interface (JNI)

riscure

- Methods in class can be marked as “native”
  - Must be provided in static library loaded with System.loadLibrary
- Native method provided through exported symbol in SO
  - Java_{package}_{class}_{method}
  - But there are more ways…
- Native function gets JNIEnv pointer
  - Basically a struct with function pointers (a few hundred) to call back into the VM

```
struct JNIEnv_ {
jint        (*GetVersion)(JNIEnv *);

jclass      (*DefineClass)(JNIEnv*, const char*, jobject, co
jclass      (*FindClass)(JNIEnv*, const char*);

jmethodID   (*FromReflectedMethod)(JNIEnv*, jobject);
jfieldID    (*FromReflectedField)(JNIEnv*, jobject);

jobject     (*ToReflectedMethod)(JNIEnv*, jclass, jmethodID,

jclass      (*GetSuperclass)(JNIEnv*, jclass);
jboolean    (*IsAssignableFrom)(JNIEnv*, jclass, jclass);

jobject     (*ToReflectedField)(JNIEnv*, jclass, jfieldID, j

jint        (*Throw)(JNIEnv*, jthrowable);
jint        (*ThrowNew)(JNIEnv *, jclass, const char *);
jthrowable  (*ExceptionOccurred)(JNIEnv*);
void        (*ExceptionDescribe)(JNIEnv*);
void        (*ExceptionClear)(JNIEnv*);
void        (*FatalError)(JNIEnv*, const char*);

jint        (*PushLocalFrame)(JNIEnv*, jint);
jobject     (*PopLocalFrame)(JNIEnv*, jobject);

jobject     (*NewGlobalRef)(JNIEnv*, jobject);
```

Java Native Interface (JNI)

riscure

```java
public class JNIActivity extends Activity {
    static {
        System.loadLibrary("myjni");
    }

    // A native method that returns a Java String to be displayed on the
    // TextView
    public native String getMessage();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView textView = new TextView(this);
        textView.setText(getMessage());
        setContentView(textView);
    }
}
```

```c
#include <jni.h>
#include "include/HelloJNI.h"

JNIEXPORT jstring JNICALL Java_com_mytest_JNIActivity_getMessage
        (JNIEnv *env, jobject thisObj) {
    return (*env)->NewStringUTF(env, "Hello from native code!");
}
```

Security Model

- App signing
  - Apps must be signed before installing.
  - Public key is part of the .apk, and is not verified.
- App sandboxing
  - Apps run under separate user/group IDs
  - Linux and SELinux enforce sandboxing between apps
- Permissions
  - Permissions control access to Android APIs (e.g. send SMS)
  - Stored in the AndroidManifest.xml

SELinux

- Additional security layer
- Mandatory Access Control (MAC) for all processes
  - Checks for every action if permission is granted or not
- Android 4.3: permissive (log violations)
- Android 4.4: partially enforced (block violations of critical processes)
- Android 5.0+ full enforcement (block all violations)
- adb shell {get|set}enforce
  - Cannot easily be set to permissive anymore

```
C:\Users\martijn\Desktop>adb shell getenforce
Enforcing
```

Keystore

- Apps must be signed
  - Modifying app requires new signature
  - Updates are **required** to be signed with the same key
- Private key stored in keystore
- Generate new key:
  - keytool -genkey -v -keystore **my-key.keystore** -alias **alias_name** -keyalg RSA -keysize 2048 -validity 10000

Android vocabulary

Name	Description
Activities	Visual screens of an application, i.e. the parts the user interacts with.
Services	Tasks running in background, e.g. receiving APDUs through HCE
Broadcast receivers	Code that listens for messages on certain events, such as receiving an SMS
Content providers	Interface that allows other apps to request data, such as account details, contacts, etc.
Intents	Messages exchanged between APIs

Static analysis of Android apps

- **META-INF/*** Certificate and signatures
- **assets/*** Files and other bundled stuff
- **lib/*** Native libraries used by the app
- **classes.dex** The actual Dalvik bytecode
- **AndroidManifest.xml** Binary manifest containing config
- **Resources.arsc** Resources such as strings, icons, etc

Manifest

- Application config
  - Application name
  - Package name
  - Version
  - allowBackup
  - debuggable
  - Activities
  - Required permissions
  - Services
  - Broadcast receivers
  - …

AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-sdk android:minSdkVersion="15" />

    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:label="@string/app_name"
        android:icon="@drawable/ic_launcher">
        <activity
            android:name="MyActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Unpacking

- Unzip?
  - classes.dex is a binary blob
    - Can be disassembled
  - resources.arsc contains resources in compiled form
  - XML (resources & manifest) stored in binary form
    - Images are optimized (and can’t be opened with normal tools)
    - 9patch files in compiled form
- Solution:
  - apktool d **example.apk** -o **example**

Decompiling an APK

riscure

- DEX → JAR → Java (with dex2jar)
  - jd-gui
  - JAD
  - Procyon
  - CFR
  - …
- DEX → Java
  - Androguard DAD
  - JADX
  - JEB (commercial)

Example

1. Let’s look at example Smali code
   - apktool d **example.apk** -o **example**
   - Inspect example/smali/MainActivity

2. Let’s look at the Java equivalent
   - Option 1: dex2jar example.apk ; jd-gui example-dex2jar.jar
   - Option 2: jadx-gui example.apk

Download URL for all exercises:

http://www.limited-entropy.com/stuff/TUD_april17.zip

Exercise #1 - RE

- Reverse engineer Exercise1.apk
  - Find out the secret password
  - Test on a phone / emulator

Repacking

1. Build
   - apktool.jar [-f] b **example** -o **example.apk.tmp**
2. Sign
   - jarsigner -verbose -keystore **sign.keystore example.apk.tmp KEYNAME**
3. Align
   - zipalign [-f] [-v 4] **example.apk.tmp example.apk**
4. Install
   - adb install -r example.apk

Example

- Let’s modify the Hello World message
  1. Change string in *MainActivity$1.smali*
  2. Build and sign new apk
  3. Test by installing it

- Some tools give nice UI for the whole process
  - APKStudio already installed in your VM!

Exercise #1b - Patching

1. Extract exercise1.apk
2. Modify password verification to always pass the check
3. Test your changes

Exercise #2: Home

1. Reverse engineer simplekeygen.apk
2. Write a key generator tool
   - Input: username
   - Output: serial number
   - Test vector: serial for *eloi* is *2bd100ae*

3. What is the serial number for the *tudelft* user?

Main techniques to protect APKs

- **Obfuscation:**

  Make RE more difficult by complicating code analysis

- **Resource protection:**

  Make extraction of resources (images, XMLs, etc.) bundled with app more difficult, e.g. by encrypting them.

- **Anti-tampering and anti-debug**

  Detect modifications (integrity checks) and environment changes (debugger, emulator, root, etc.)

Some examples - Obfuscation

```
private static String T5OatD;

static {
    ClassManager.T5OatD = q18dwJ.T5OatD("傾笑");
}
```

Class and identifier renaming

Encrypted strings

```
public Object invoke(Object arg13, Method arg14, Object[] arg15) {
    if(arg14.getDeclaringClass() == Object.class) {
        return arg14.invoke(this, arg15);
    }
```

Reflection API

```
public static String a()
{
  return (String)AuroraBridge.a(393, new Object[0]);
}

public static String a(Object paramObject)
{
  try
  {
    Object localObject = AuroraBridge.a(441, new Object[] { paramObject });
    return (String)localObject;
  }
  catch (Exception localException)
  {
    throw AuroraBridge.a(localException);
  }
}
```

(native) proxy classes

Are these techniques infallible?

riscure

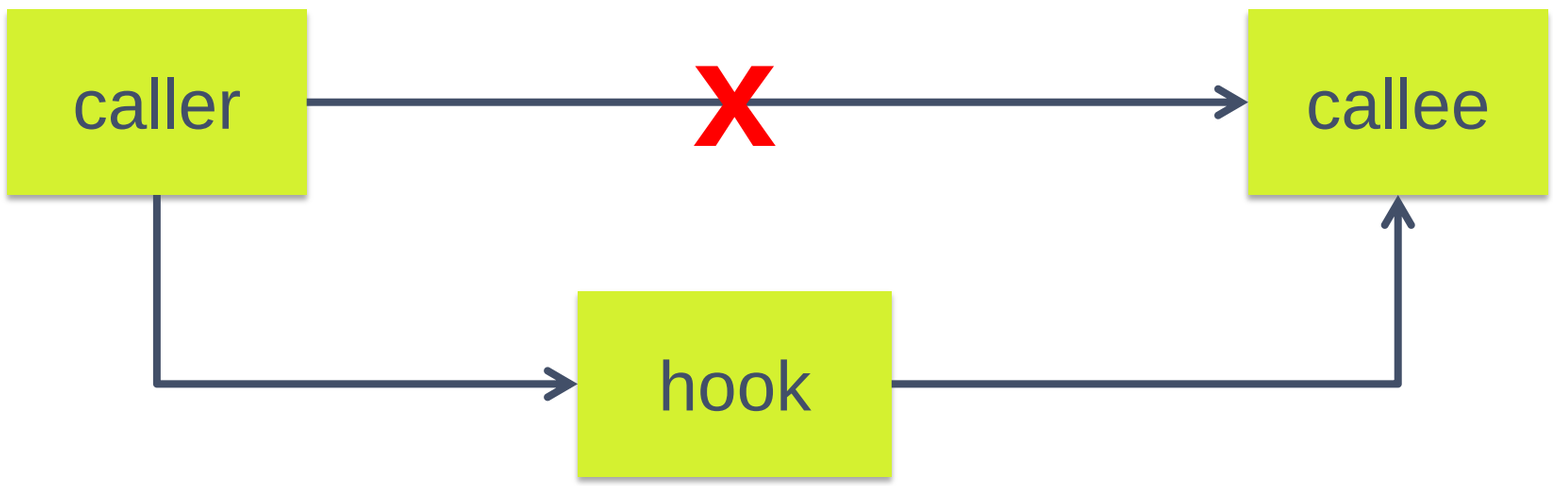
- Obviously not: with enough effort they can be bypassed
  - Slowing down attacker still valuable!
- Obfuscation protects mainly against static analysis
  - Dynamic analysis can give a lot of info!
- Anti-tamper/debug can be patched or worked around
  - There might be many checks to bypass/patched
  - You might be better off looking for alternatives (e.g. hooking instead of debugging)

Break time!

Hooking Android apps with Frida

Hooking?

In computer programming, the term hooking covers a range of techniques used to alter or augment the behavior of an operating system, of applications, or of other software components by intercepting function calls or messages or events passed between software components. Code that handles such intercepted function calls, events or messages is called a "hook".



What can we do with hooking?

riscure

1. Inspect and log calls

2. Prevent calls
   - E.g. skip setting a flag that indicates security violation

3. Modify parameters
   - E.g. redirect communications to a malicious server

4. Modify return values
   - E.g. always return *true* after password check

5. Replace a function
   - E.g. hide suspicious files from the application

Frida hooking framework

- Cross-platform
  - Windows, Linux, Android, iOS, OSX with the same API!
- Injects JavaScript engine into target process
  - Instrumentation written in JavaScript
  - Allows hooking native and Java side on Android
- Control logic can be written in many languages
  - We'll use Python
- Several other frameworks available
  - We focus on this one for the exercises

Frida on Android

- Installed with *sudo pip install frida*
  - See http://www.frida.re/docs/installation/ for more info
  - frida-server provided in zip file

- Need to push frida-server to target

```
adb push frida-server /data/local/tmp
adb shell “chmod 755 /data/local/tmp/frida-server
adb shell “su –c /data/local/tmp/frida-server &”
```

- Now run the following command to list processes:

```
frida-ps -U
```

Example native code

riscure

```c
#include <stdio.h>
#include <unistd.h>

int main(int ac, char **av){
  unsigned char buf[32];
  int i;
  FILE *fd;
  while(1){
    fd = fopen("/dev/urandom", "r");
    fread(buf, 32, 1, fd);
    for(i=0;i<32;i++){
      printf("%.2x ", buf[i]);
    }
    printf("\n");
    fclose(fd);
    sleep(1);
  }
  return 0;
}
```

Want to fix this random

Python code (attaching)

Import necessary modules

```
import frida
import sys
```

Connect to ‘test’ on Android device

```
if len(sys.argv) > 1 and sys.argv[1] == “usb":
  session = frida.get_usb_device(1).attach("test") # 1 second timeout
else:
  session = frida.attach("test")
```

Load script from fopen.js

```
script = session.create_script(open("fopen.js").read())
```

Handle messages from script, if any

```
def on_message(message, data):
    print(message)


script.on('message', on_message)
```

Load script onto target, and wait for input on stdin

```
script.load()
sys.stdin.read()
```

JavaScript side (1)

- Find pointer to *fopen* and *fread*

```
var libc = "libc-2.21.so"
var fopenPtr = Module.findExportByName(libc, "fopen");
if (fopenPtr == null){
  libc = "libc.so"
  fopenPtr = Module.findExportByName(libc, "fopen");
}

var freadPtr = Module.findExportByName(libc, "fread");
```

JavaScript side (2)

- Replace *fopen* with new implementation

```
var rndFd;
var fopen = new NativeFunction(fopenPtr, 'pointer',
['pointer', 'pointer']);
var fopenCallback = new NativeCallback(function (pathPtr,
modePtr) {
  var mode = Memory.readUtf8String(modePtr);
  var path = Memory.readUtf8String(pathPtr);

    var fd = fopen(pathPtr, modePtr);

    if (mode == "r" && path == "/dev/urandom"){
    console.log("Got fd for /dev/urandom: " + fd);
    rndFd = fd;
  }
    return fd;
}, 'pointer', ['pointer', 'pointer']);
Interceptor.replace(fopenPtr, fopenCallback);
```

Call original *fopen*

Cache /dev/urandom file descriptor

JavaScript side (2)

- Replace *fread* with new implementation

```
var fread = new NativeFunction(freadPtr, 'uint',
['pointer', 'uint', 'uint', 'pointer']);
Interceptor.replace(freadPtr, new NativeCallback(function
(dataPtr, size, numElem, filePtr){
  if (filePtr.equals(rndFd)){
    console.log("Found fread from /dev/urandom!!");
    console.log("Size: " + size );
    console.log("Elems: " + numElem );
    arr = new Array(size*numElem);
    for(var i=0;i<size*numElem;i++)
    arr[i] = 0;
    Memory.writeByteArray(dataPtr, arr);
    return numElem;
  } else {
    console.log("Calling original fread on " + filePtr);
    return fread(dataPtr, size, numElem, filePtr);
  }
}, 'uint', ['pointer', 'uint', 'uint', 'pointer']));
```

Check file descriptor

Fill buffer with custom data

DEMO – Frida Native hooking

Hooking the Java side with Frida

- Faking registration on exercise1.apk
  1. Hook *check* method
  2. Always return *true*

```
'use strict';

//Make sure we are in a dalvik context
Java.perform(function () {
  // Get a handle to the class
  const MainActivity =
Java.use('com.riscure.exercise1.MainActivity');
  //Replace implementation
  MainActivity.check.implementation = function (){
    return true;
  };
});
```

DEMO – Frida Java hooking

Other hooking options

- Hooking can be done at many levels
  - Interposing a library with *LD_PRELOAD*
  - Using a *shim* library
  - Injecting code into application (a-la Frida)
  - Injecting code into the kernel (e.g. syscall hooking)

- Which one to use depends on your target
  - Java code vs native code?
  - Protection against user-space hooking?

- Deeper hooks → more difficult to detect
  - But also trickier to implement usually!

Summary

- Several techniques to analyze Android apps
  - Disassembly (smali) and decompilation (Java)
  - Modification and repackaging
  - Dynamic analysis through hooking

- Briefly discussed protection techniques
  - Obfuscation
  - Anti-tampering
  - Anti-analysis

- Protection deters attackers, doesn’t stop them
  - Plan for compromise!
  - Get your apps tested if security is important

riscure

Eloi Sanfelix (@esanfelix)
eloi@riscure.com