**Root Detection**

* Goal: to make it a **bit more difficult to run the app on a rooted device**, which in turn impedes some tools and techniques reverse engineers like to use. As with most other defenses, root detection is **not highly effective** on its own, but having some root checks sprinkled throughout the app can improve the effectiveness of the overall anti-tampering scheme.
* **Root detection** : verifying whether the device is a stock Android build or a custom build.
* **Common Root Detection Methods:**
* SafetyNet : is an Android API that creates a profile of the device using software and hardware information. This profile is then compared against a list of white-listed device models that have passed Android compatibility testing.

SafetyNet is not well documented, and may change at any time: When you call this API, the service downloads a binary package containing the device validation code from Google, which is then dynamically executed using reflection. An analysis showed that the checks performed by SafetyNet also attempt to detect whether the device is rooted, although it is unclear how exactly this is determined.

To use the API, an app may the SafetyNetApi.attest() method with returns a ***JWS*** message with the Attestation Result, and then check the following fields:

**ctsProfileMatch**: Of "true", the device profile matches one of Google's listed devices that have passed Android compatibility testing.

**basicIntegrity**: Of "true", The device running the app likely wasn't tampered with.

The attestation result looks as follows.

{

"nonce": "R2Rra24fVm5xa2Mg",

"timestampMs": 9860437986543,

"apkPackageName": "com.package.name.of.requesting.app",

"apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the

certificate used to sign requesting app"],

"apkDigestSha256": "base64 encoded, SHA-256 hash of the app's APK",

**"ctsProfileMatch": true,**

**"basicIntegrity": true,**

}

**JWS** *is a way to authenticate (but not necessarily encrypt) information in a highly serializable, machine-readable format. That means that it is information, along with proof that the information hasn't changed since being signed. It can be used for sending information from one web site to another, and is especially aimed at communications on the web. It even contains a compact form optimized for applications like URI query parameters.*

* Programmatic Detection:  the most widely used method is checking for files typically found on rooted devices, such as package files of common rooting apps and associated files and directories.

Such as

/system/app/**Superuser**.apk

/system/etc/init.d/99SuperSUDaemon

/dev/com.koushikdutta.superuser.daemon/

/system/xbin/**daemonsu**

* Detection code also often **looks for binaries that are usually installed once a device is rooted**. Examples include checking for the presence of busybox or attempting to open **the su binary** at different locations:

/system/xbin/busybox

/sbin/**su**

/system/bin/**su**

/system/xbin/**su**

/data/local/**su**

/data/local/xbin/**su**

* **checking whether su is in PATH also works**:

public static boolean checkRoot(){

for(String pathDir : System.getenv("PATH").split(":")){

if(new File(pathDir, "su").exists()) {

return true;

}

}

return false;

}

* **JNI** uses the stat system call to retrieve information about a file and returns 1 if the file exists

jboolean Java\_com\_example\_statfile(JNIEnv \* env, jobject this, jstring filepath) {

jboolean fileExists = 0;

jboolean isCopy;

const char \* path = (\*env)->GetStringUTFChars(env, filepath, &isCopy);

struct stat fileattrib;

if (stat(path, &fileattrib) < 0) {

\_\_android\_log\_print(ANDROID\_LOG\_DEBUG, DEBUG\_TAG, "NATIVE: stat error: [%s]", strerror(errno));

} else

{

\_\_android\_log\_print(ANDROID\_LOG\_DEBUG, DEBUG\_TAG, "NATIVE: stat success, access perms: [%d]", fileattrib.st\_mode);

return 1;

}

return 0;

}

* **Checking running processes:**

Supersu - by far the most popular rooting tool - runs an **authentication** daemon named **daemonsu**, so the presence of this process is another sign of a rooted device. Running processes can be enumerated through

ActivityManager.getRunningAppProcesses() and manager.getRunningServices() APIs, the **ps** command, or walking through the /proc directory

public boolean checkRunningProcesses() {

boolean returnValue = false;

// Get currently running application processes

List<RunningServiceInfo> list = manager.getRunningServices(300);

if(list != null){

String tempName;

for(int i=0;i<list.size();++i){

tempName = list.get(i).process;

if(tempName.contains("supersu") || tempName.contains("**superuser**")){

returnValue = true;

}

}

}

return returnValue;

}

* **Checking installed app packages:**

**The Android package manager can be used to obtain a list of installed packages. The following package names belong to popular rooting tools:**

**com.thirdparty.superuser**

**eu.chainfire.supersu**

**com.noshufou.android.su**

**com.koushikdutta.superuser**

**com.zachspong.temprootremovejb**

**com.ramdroid.appquarantine**

* **Checking for custom Android builds:**

Besides checking whether the device is rooted, it is also helpful to check for signs of test builds and custom ROMs. One method of doing this is checking whether the BUILD tag contains test-keys, which normally [indicates a custom Android image](http://resources.infosecinstitute.com/android-hacking-security-part-8-root-detection-evasion/). This can be [checked as follows](https://www.joeyconway.com/blog/2014/03/29/android-detect-root-access-from-inside-an-app/):

private boolean isTestKeyBuild()

{

String str = Build.TAGS;

if ((str != null) && (str.contains("test-keys")));

for (int i = 1; ; i = 0)

return i;

}

##### Bypassing Root Detection: disable the root detection mechanisms.

* You can use a number of techniques to bypass these checks such as :
* It is possible to easily bypass the mechanisms using standard tools such as **RootCloak.**
* **Renaming binaries**. For example, in some cases simply renaming the "su" binary to something else is enough to defeat root detection (try not to break your environment though!).
* **Unmounting /proc to prevent reading of process lists** etc. Sometimes, proc being unavailable is enough to bypass such checks.
* **Using Frida or Xposed to hook APIs on the Java and native layers**. By doing this, you can hide files and processes, hide the actual content of files, or return all kinds of bogus values the app requests;
* **Hooking low-level APIs using Kernel modules.**
* **Patching the app to remove the checks.**

##### 

**References:**

[1] <https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05j-Testing-Resiliency-Against-Reverse-Engineering.md#safetynet>