Challenges in Android Supply Chain Analysis

Łukasz Siewierski (@maldr0id)
Reverse Engineer, Android Security (Google)
Agenda

• What does an Android device and system updates go through before its first public sale?

• What are technical challenges in analysing Android system images?

• Case studies
The journey of an Android device
The journey of an Android device
### Approval process for Android devices

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTS (Compatibility Test Suite)</td>
<td>Ensuring compatibility with AOSP</td>
</tr>
<tr>
<td>GTS (GMS Requirements Test Suite)</td>
<td>Requirements for any devices that want to license Google apps</td>
</tr>
<tr>
<td>VTS (Vendor Test Suite)</td>
<td>Compatibility with the Hardware Abstraction Layer (HAL)</td>
</tr>
<tr>
<td>STS (Security Test Suite)</td>
<td>Checks if security patches have been applied correctly</td>
</tr>
<tr>
<td>BTS (Build Test Suite)</td>
<td>Security review for malware and other harmful behaviors in binaries / framework</td>
</tr>
</tbody>
</table>
List of requirements that must be met in order for devices to be compatible with the latest version of Android.

For example section 9 deals with “Security Model Compatibility” and contains subsections relating to:

- Permissions
- Premium SMS warning
- Security Features (e.g. SELinux)
- Data Storage Encryption
- Automotive Vehicle System Isolation
Android system image analysis challenges
Case study #1

Device monitoring and dynamic analysis challenges
It started with an application

```xml
<receiver android:name="com.[redacted].receiver.AppMonitorReceiver">
  <intent-filter>
    <action android:name="com.[redacted].appmonitor.app_onCreate"/>
    <action android:name="com.[redacted].appmonitor.app_onResume"/>
    <action android:name="com.[redacted].appmonitor.load_url"/>
  </intent-filter>
</receiver>
```

Odd intent names?

```java
public void onReceive(android.content.Context context, android.content.Intent intent) {
    String action = intent.getAction();
    if (action.equals(this.load_url_intent)) {
        addsURLAndPackNameToDatabase(context, intent);
    }
}
```

Expects two extra fields

```java
public void addsURLAndPackNameToDatabase(android.content.Context context, android.content.Intent intent) {
    String url = intent.getStringExtra("url");
    String packname = intent.getStringExtra("packname");
    addURLInfoToDatabase(context, url, packname);
}
```

Adds data to the database
Additional (unused) method in the AOSP Activity class

private void sendNewAppBroadcast() {
    String lastpkg = System.getString(this.getContentResolver(), "lastpkg");
    String curpkg = this.mActivityInfo.applicationInfo.packageName;
    if (lastpkg == null || !lastpkg.equals(curpkg)) {
        Intent it = new Intent();
        it.setAction("com.[redacted].app_onResume");
        it.putExtra("packname", curpkg);
        this.sendBroadcast(it);
    }
}

public void loadUrl(String url) {
    this.checkThread();
    Log.d("WebView", "loadUrl=" + url);
    this.mProvider.loadUrl(url);
    Application initialApplication = AppGlobals.getInitialApplication();
    if(initialApplication != null && (URLUtil.isNetworkUrl(url))) {
        Intent it = new Intent();
        it.setAction("com.[redacted].load_url");
        it.putExtra("url", url);
        it.putExtra("packname", initialApplication.getPackageName());
        initialApplication.getApplicationContext().sendBroadcast(it);
    }
}
We worked with the OEM to provide a system update which removes the additional code.

To protect users before they get the system update, the app that gathers information is disabled by Play Protect.
Dynamic analysis - challenges

• The apps need specific AOSP modifications in order to work
• The apps need specific devices / drivers in order to work
• The apps that you’re trying to install are already on the device (see below)

$ adb install com.android.systemui.apk
adb: failed to install com.android.systemui.apk: Failure [INSTALL_FAILED_VERSION_DOWNGRADE]

$ adb install com.android.systemui.apk
Failure [INSTALL_FAILEDOLDER_SDK]

$ adb install com.android.systemui.apk
adb: failed to install com.android.systemui.apk: Failure [INSTALL_FAILED_UPDATE_INCOMPATIBLE: Package com.android.systemui signatures do not match the previously installed version; ignoring!]
Is there a way to make dynamic analysis work?

You have to use some of the same methods OEMs use:

- Have your own modified Android image on the emulator
- Sign apps with your own “platform” key
- Install them in /system by moving the APK files to the /system partition

However, if the Android framework is modified you don’t have enough luck and you have to resort to static analysis, which leads us to...
Case study #2

Triada and the complexity of static analysis
Triada history

- Triada rooting Trojan was first described by Kaspersky in March 2016

- System level backdoor in summer 2017 (described by Dr Web in July 2017)

  Since then we worked with the OEMs to remove Triada from all the devices, both old and new

Triada, from the early rooting trojan days was investing heavily in code injection and the system-level backdoor pushed it even further...
Backdooring the AOSP log function

Triada backdoored the log function to perform code injects

This line of code was added

Code was injected to com.android.systemui in order to have the GET_REAL_TASKS permission

Code was also injected to com.android.vending to allow for these operations:

1. 下载请求  1. download request
2. 下载结果  2. download result
3. 安装请求  3. install request (uses real, unpopular Google Play package names)
4. 安装结果  4. installation result
5. 激活请求  5. activation request
6. 激活结果  6. activation result
7. 拉活请求  7. pull request
8. 拉活结果  8. pull the results
9. 卸载请求  9. uninstall request
10. 卸载结果 10. uninstall result
Complex communication mechanisms

Play
- Code injection

System UI
- Code injection

Installed app
- Socket intent

Framework backdoor

Java properties

Log lines

C&C
We worked with all the affected OEMs to provide system updates which remove Triada.
Aside: how did it get on the device?

OEM

(the company that sells the device)

sends a whole system image

3rd party vendors

(companies that provide additional features)

sends back the system image…

… not mentioning Triada

adds additional features to the system image…

… including Triada
How to make static analysis work?

- Take a look at the whole system image (including binaries, services and non-standard file objects)
- Take a look at the framework files - they may have additional code
- Try to understand the ecosystem of a system image holistically - which process interacts with which app and what are the SELinux rules, which brings us to...
Case study #3

App update framework and sometimes things aren’t what they seem
App update framework

Binary running as root on the system image in the /bin directory

Executes in several stages (original naming):

0. check system directory
1. check debug status
2. check if rooted
3. register signal handler and do miscellaneous work
4. create communicate fifo
5. check main imei status
6. check dual sim status
7. add predefined system task
8. enter main loop

/data/[redacted]/fifo_in

binary

/data/[redacted]/fifo_out
Two ways to pass the commands

Passed through the \texttt{fifo\_in} file:

- run the argument as a shell script
- kill a specific process by name or pid
- execute arguments as a command
- prints arguments to \texttt{fifo\_out}
- downloads a file

Passed as an argument:

- remount the /system partition as rw
- download a shell script file from the C&C and executes it
- upload any file to the remote server
- print the version of the binary
- execute a binary given as an argument
Dialer app creates commands

```java
public static boolean handleCode(android.content.Context context, String code) {
    String command = "0";
    if (code.equals("*#9381#")) { command = "#update{-g} [...] -upc\n"; }
    if (code.equals("*#9382#")) { command = "#update{-g} [...] -upi\n"; }

    java.io.File fifo_in = new java.io.File("/data/[redacted]/fifo_in");
    java.io.FileOutputStream task_pool = new java.io.FileOutputStream(fifo_in);
    byte[] to_write = command.getBytes();
    task_pool.write(to_write, 0, to_write.length);
    task_pool.flush();
    task_pool.close();
    return true;
}
```
... but it cannot be abused

Untrusted app cannot write into the fifo_in file due to SELinux

Untrusted app cannot write into the fifo_in file due to SELinux

And the binary drops privileges...

```c
if ( calling_uid ) {
    printf("set uid to %d\n", calling_uid);
    setuid(calling_uid);
}

if ( calling_gid ) {
    printf("set gid to %d\n", calling_gid);
    setgid(calling_gid);
}
```
We worked with the OEM to audit their security configurations and make sure it cannot be abused.
Can build fingerprint by itself be used to identify a device?

<table>
<thead>
<tr>
<th>Build Fingerprint</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMSUNG/T805S</td>
<td>T805S:4.4.2/KOT49H/20170918.165122:user/release-keys</td>
</tr>
<tr>
<td>Samsung/T1000</td>
<td>T1000:4.4.2/KOT49H/20180730.204004:user/release-keys</td>
</tr>
<tr>
<td>SAMSUNG/T9500</td>
<td>SAMSUNG/T950:4.4.2/KOT49H/20150317.141621:user/release-keys</td>
</tr>
<tr>
<td>SAMSUNG/T800</td>
<td>T800:4.4.2/KOT49H/20170918.165122:user/release-keys</td>
</tr>
<tr>
<td>Samsung/T950</td>
<td>SAMSUNG/T950:4.4.2/KOT49H/20150317.141621:user/release-keys</td>
</tr>
<tr>
<td>SAMSUNG/T950S</td>
<td>SAMSUNG/T950S:4.4.2/KOT49H/20170918.165122:user/release-keys</td>
</tr>
<tr>
<td>SAMSUNG/T805S</td>
<td>T805S:4.4.2/KOT49H/20180730.204004:user/release-keys</td>
</tr>
<tr>
<td>Samsung/T10</td>
<td>T10:4.4.2/KOT49H/20180130.175513:user/release-keys</td>
</tr>
<tr>
<td>SAMSUNG/SAMSUNG</td>
<td>SAMSUNG/SAMSUNG:4.4.2/KOT49H/20171207.183925:user/release-keys</td>
</tr>
<tr>
<td>Samsung/SAMSUNG</td>
<td>SAMSUNG/SAMSUNG:4.4.2/KOT49H/20171207.183925:user/release-keys</td>
</tr>
</tbody>
</table>

All of these **build fingerprints** are for non-CTS, non-Samsung devices.
Are all the apps on /system preinstalled?

Rooting trojans can remount the /system partition and move there to avoid uninstallation.

Although **Verified Boot** prevents it on newer Android devices.
Summary
Summary: preinstalled statistics for 2019

- 2.5+ billion devices protected
- 3+ million preinstalled applications scanned
- 50+ thousand system images scanned
Researchers:

• We need more researchers working in the preinstalled app space.

• Understanding a few key differences when analyzing pre-installed apps versus user-space apps can help your analysis be more efficient.

• The Android ecosystem is vast with a diversity of OEMs & their customizations. This comes with new and exciting features for users, but also new and exciting challenges for security researchers.
Everyone else:

• Build Test Suite is used to scan all the system images for any preinstalled malware, including system image updates
• We are also using in-the-wild monitoring to find new malware, including preinstalled ones
• Google Play Protect alerts the user of any malware and removes or disables them
• We are also working with the OEMs to provide system updates which remove preinstalled malware
• Take a look at the Android Enterprise website at android.com/enterprise
Thank you!