- http://lepouvoirclapratique.blogspot.fr/
- Cédric BERTRAND
- Novembre 2012

Pentest d'une application Android

Sommaire

Table des illustrations

Documents de référence

[FOUNDSTONE] Penetration Testing Android Applications (Kunjan SHAH)
[MISC] Méthodologie de pentest pour applications Android (Nassim ABBAOUI)
[MSE] Menaces et failles du système Android (Carolo Critini)
[BLOG] Analyse de malwares sous Android (Cédric BERTRAND)
[SYMANTEC] Android Application Security Assessments

1. Evaluer la sécurité d'une application Android

Pour évaluer la sécurité d'une application mobile, les méthodes utilisées restent assez similaires à celles utilisées sur l'informatique classique, à savoir :

- Audit du code-source de l'application
- Pentest & audit de sécurité de l'application (capture du trafic réseau, analyse dynamique, etc.)

Il va donc s'agir d'évaluer la sécurité d'une application en répondant à certaines questions telles que[1] :

- Confidentialité des échanges
  - De quelle façon est sécurisée la transmission d'information sur le réseau?
  - Quel chiffrement l'application met-elle en œuvre ? Celui est-il adapté ?
  - Quel sont les types de réseaux utilisé par l'application ? Quel est le degré de confidentialité de chacun de ces réseaux ?
- Intégrité des échanges
  - Depuis le téléphone
    - Est-il possible d'intercepter les communications depuis le téléphone?
      - Risques ?
    - Est-il possible de modifier les communications depuis le téléphone?
      - Risques ?
  - Depuis un élément réseaux
    - Est-il possible d'intercepter les communications?
      - Risques ?
    - Est-il possible de modifier les communications?
      - Risques ?
- Etc.

C'est l'objet de ce document qui décrit une méthodologie afin de réaliser un audit de sécurité et un pentest d'une application Android. Nous verrons en particulier comment capturer le trafic, décompiler une application Android et quelles sont les vulnérabilités à rechercher.

[1] http://wiki.frandroid.com/wiki/Penser_la_s%C3%A9curit%C3%A9_de_ses_applications_mobiles

2. Pentest d'une application Android

2.1. Démarche vis-à-vis d'un pentest d'application « classique »

Vis à vis d'un pentest dit « classique », la démarche est en général la même : récupération d'informations, recherche en vulnérabilités et scénarios d'exploitation. Ce qui varie avec le pentest d'applications Android est surtout l'environnement de test qui nécessite un device soit réel (téléphone), soit virtuel (émulateur).

Nous constaterons au cours de ce document que la sécurité d'une application Android, dépend et de l'application elle-même et du *backend*[2] utilisé, dans le cas où elle doit communiquer avec un serveur web par exemple.

2.2. L'environnement de test

L'émulateur fourni par Google permet en général de répondre à la plupart des besoins. Par contre l'inconvénient de l'émulateur est la difficulté d'obtenir le Google Play Store. Son installation sur un émulateur pouvant se révéler fastidieuse, la récupération de l'application par un téléphone se révèle plus simple.

Concernant les avantages d'un émulateur vs un téléphone, nous avons : le root d'office (réalisé par défaut sous l'émulateur), la simplification de la capture du trafic réseau, et enfin la modification de certains paramètres censés être fixe (IMEI, IMSI).

2.3. Les outils

De nombreux outils existent afin d'aider à l'analyse d'une application Android : certains permettent de contrôler le device, d'autres de simuler un appel / réception de messages par exemple.

2.3.1. ADB

Cet outil en ligne de commandes permet de contrôler un device Android (via USB) ou avec une instance de l'émulateur. Situé dans le répertoire <**$SDK/platform-tools/adb**>, il se compose de 3 parties : un daemon (qui fonctionne sur le device Android), un client sur la machine de tests ainsi que le serveur chargé de la communication. Parmi les commandes les plus utiles, nous avons :

- Consulter la liste des périphériques actifs : **adb devices**

```
C:\Program Files\Android\android-sdk\platform-tools>adb devices
List of devices attached
emulator-5554   device
```

Figure 1 Liste des périphériques actifs avec l'outil ADB

- Ouvrir un shell : **adb shell**

[2] http://fr.wikipedia.org/wiki/Backend

```
C:\Program Files\Android\android-sdk\platform-tools>adb shell
# cd data
cd data
# ls
```

Figure 2 Ouverture d'un shell avec l'outil ADB

- Installer une application (apk) : **adb install <path_de_l'application>**

```
C:\Program Files\Android\android-sdk\platform-tools>adb install c:\Samples\Geini
mi_a.apk.APK
65 KB/s (2808157 bytes in 41.640s)
        pkg: /data/local/tmp/Geinimi_a.apk.APK
Success
```

- Copier un fichier du téléphone à son ordinateur (et inversement) : **adb push <local> <remote> / adb pull <remote> <local>**

```
C:\Program Files\Android\android-sdk\platform-tools>adb push "c:\Tools Android\P
ortSwiggerCA.cer" /sdcard/
3 KB/s (894 bytes in 0.281s)
```

Pour connaitre la liste des autres commandes, vous pouvez consulter le lien suivant.

2.3.2. DDMS (Dalvik Debug Monitor Server)

Cet outil est un débuggeur fournit avec Android, et permet outre le débogage de l'application, la simulation et la réception d'appels / SMS ainsi que la géolocalisation. Il est disponible dans le répertoire <**$SDK/tools/ddms>** et peut être rattaché à un device réel ou à l'émulateur.

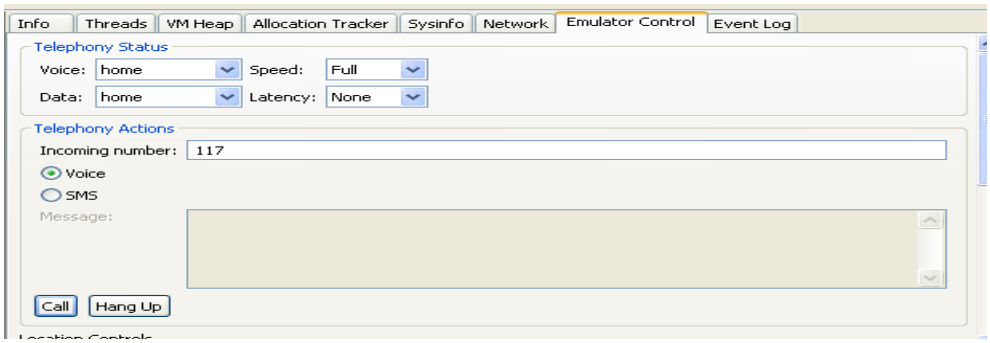


Figure 3 Simulation d'appels avec DDMS

Cet outil permet de connaître toutes les actions effectuées sur le device. On peut par exemple l'utiliser pour récupérer une application installée à partir d'un site web, savoir les actions réalisées, les fichiers copiés, etc. Exemple avec **X-Ray** [3] qui est une application permettant de scanner les vulnérabilités d'un terminal Android.

[3] http://www.xray.io/

Figure 4 Téléchargement d'une application depuis le site web

Une fois le téléchargement et l'installation de l'application réalisée à partir du site web, **DDMS** permet de savoir où l'application a été téléchargée et de la récupérer en conséquence.

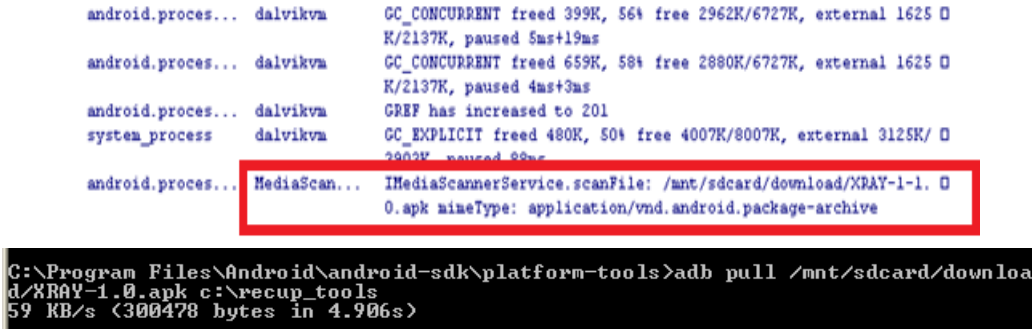


Figure 5 Récupération de l'application

2.3.3. SQLite

En général, les données sous Android sont stockées dans des bases de données au format *SQLite*[4] (extension .db ou .sqlite). Pour les consulter, on peut utiliser l'outil **sqlite** fourni avec le SDK. Pour plus d'informations sur les différentes options de stockage dans Android, vous pouvez consulter ce lien.

Un exemple pour consulter le journal d'appel du téléphone :

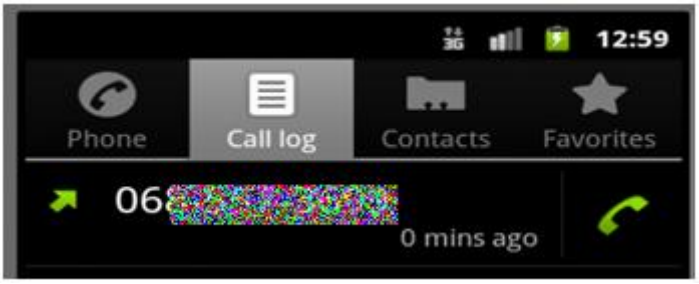


Figure 6 Consultation des appels

Ces données se trouvent dans le répertoire **</data/data/com.android.providers.telephony/databases/contacts2.db>.** On ouvre la base de données avec sqlite3.

Figure 7 Ouverture de la base de données

[4] http://fr.wikipedia.org/wiki/SQLite

Puis on utilise la commande « .table » pour connaitre les tables disponibles et la commande « .schema » pour connaitre la structure d'une table.

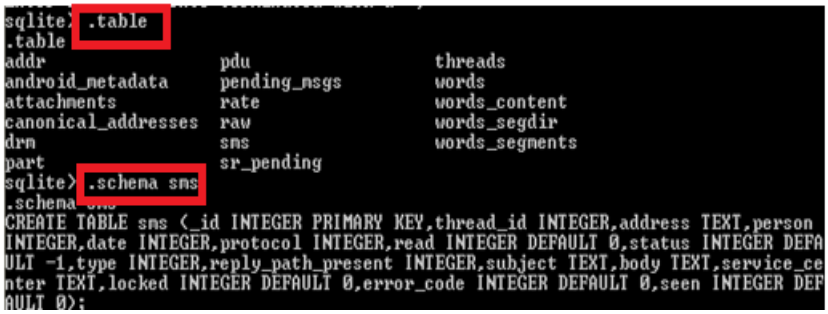


Figure 8 Connaitre les tables et leur structure

Enfin la commande « select » permet d'interroger la base de données.

Figure 9 Consultation du journal d'appel

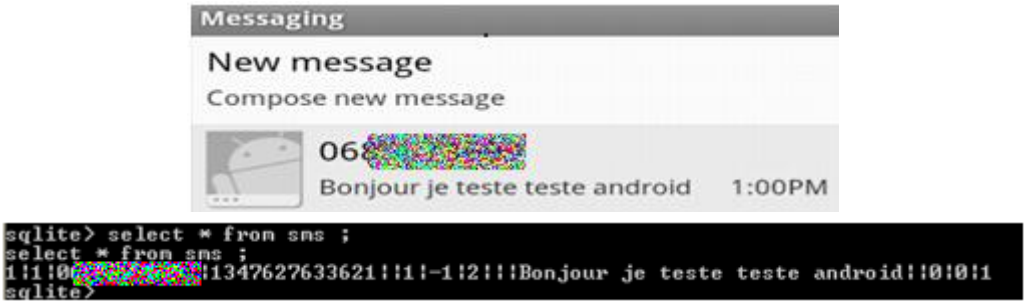
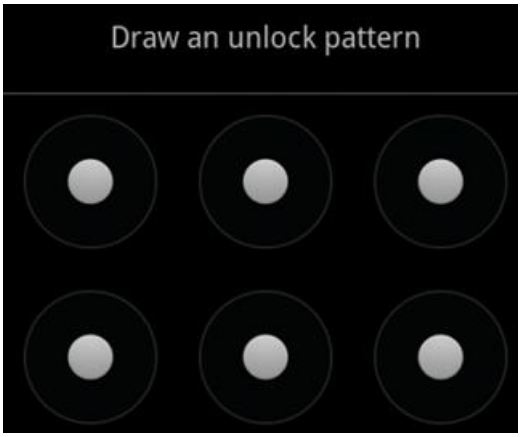


Figure 10 Consultation des sms

En annexe 1, vous trouverez une liste des emplacements des données stockées sous Android.

En utilisant **SQLite** et **DDMS**, il est possible de consulter les données écrites et lues par une application. La capture suivante montre par exemple la mise en place d'un schéma de sécurité afin de verrouiller le terminal:



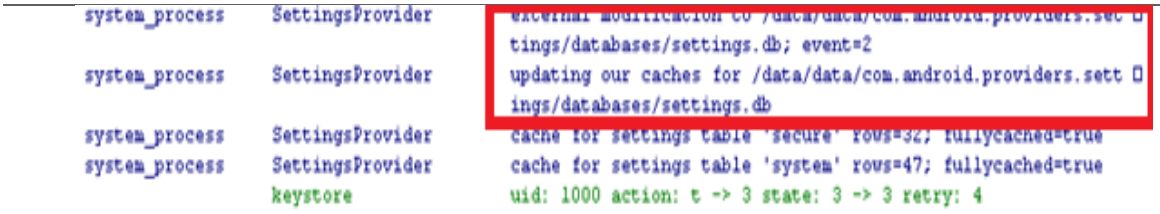


Figure 11 Mise à jour du schéma de sécurité sous Android

L'outil **DDMS**, nous indique que le schéma de sécurité a été écrit dans la base <**/data/data/com.android.providers.settings/databases/settings.db**>

2.4. L'analyse du trafic

La capture du trafic est un des points essentiels de l'analyse de la sécurité d'une application. Cela permet d'analyser les informations qui sont échangées, si celles-ci sont chiffrées, le serveur de communication, etc. Pour effectuer ce type d'actions, il va falloir faire passer le trafic par un *proxy*[5].

2.4.1. Capturer le trafic

POur utiliser un proxy web avec l'émulateur, on lance la commande <**–http-proxy**>.

Figure 12 Lancer un proxy avec l'émulateur Android

En mettant en écoute un proxy web (par exemple **Burp**[6], **Paros proxy**[7], etc.) sur le port d'écoute, on peut ainsi intercepter les communications.

Figure 13 Exemple d'interception du trafic

Le souci avec cette méthode est que seul le trafic provenant du navigateur (et non des autres applications) est intercepté. Une méthode pour corriger ceci, est d'utiliser l'application ProxyDroid qui gère plusieurs protocoles (http, https, Socks4 & 5) et permet de configurer le

[5] http://fr.wikipedia.org/wiki/Mandataire_%28informatique%29
[6] http://portswigger.net/burp/proxy.html
[7] http://www.parosproxy.org/

proxy pour une application spécifique. Attention sur un terminal, cette application nécessite d'être root afin d'installer la suite **BusyBox**[8] **(**modification des règles *iptables*[9]).

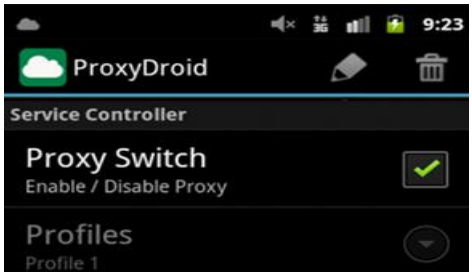


Figure 14 ProxyDroid

Malheureusement en cas de connexions HTTPS, cette méthode ne va pas s'avérer suffisante.

2.4.2. Capturer le trafic chiffré

Si l'application tente d'établir des connexions HTTPS, nous allons avoir un certain nombre d'alertes qui vont être levées ce qui est normal car l'autorité de certification ne sera pas reconnue sur le système. C'est d'ailleurs bon signe car cela indique qu'est vérifiée la validité du certificat. Si ce n'est pas le cas, c'est une vulnérabilité…

Figure 15 Message d'erreur lors d'une connexion https (site twitter)

Il va donc falloir ajouter l'autorité de certification du proxy au *TrustStore d'Android*. La manipulation diffère selon que l'on soit sur Android 4.0 ou une version inférieure.

Avant d'ajouter le certificat, un problème que l'on va rencontrer vient du fait que **Burp** génère un certificat par *ip* et non par *host*. Avant de commencer la manipulation décrite dans les paragraphes suivants, il faut tout d'abord indiquer à **Burp**[10] de générer un certificat pour un *hostname* spécifique. Pour Burp[11], ceci s'effectue dans les options du proxy.

[8] https://play.google.com/store/apps/details?id=stericson.busybox&hl=fr
[9] http://fr.wikipedia.org/wiki/Iptables
[10] http://www.crazyws.fr/android/burp-proxy-crack-le-ssl-sur-android-YJYSR.html
[11] http://www.portswigger.net/burp/help/proxy_options.html#listeners_cert

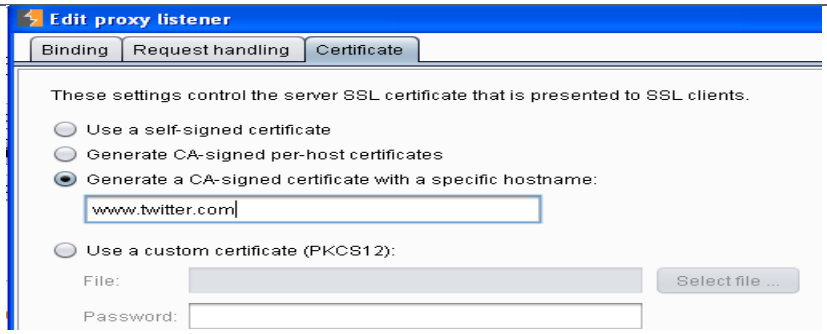


Figure 16 Changement de la méthode de génération des certificats sous Burp

Ceci réalisé, l'étape suivante est de récupérer un certificat afin de l'ajouter à Android. Pour cela, on ouvre son navigateur, on va sur un site qui utilise une connexion chiffrée (exemple : site de Twitter), et on regarde les détails du certificat.

Figure 17 Détails du certificat

Dans l'onglet détails, on exporte le certificat racine (portSwigger) au format X509. Puis on renomme le certificat en .crt (format accepté par Android)

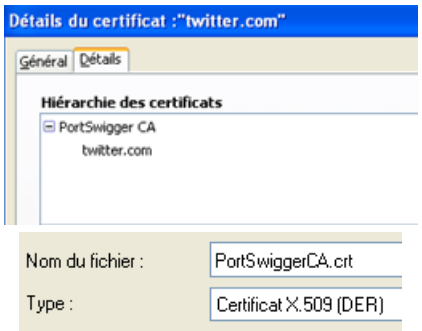


Figure 18 Exportation du certificat

Une fois le certificat exporté, on va devoir l'ajouter au Truststore d'Android. avec l'outil **ADB**.

```
C:\Program Files\Android\android-sdk\platform-tools>adb push "c:\Tools Android\P
ortSwiggerCA2.crt" /sdcard/
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
5 KB/s (712 bytes in 0.125s)
```

Figure 19 Copie du certificat sur la carte SD

Puis dans le device Android, on va dans <**settings** -> **sécurité** -> **credential storage** -> **Install from SD card**>.

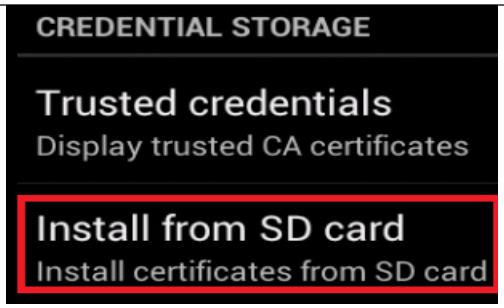
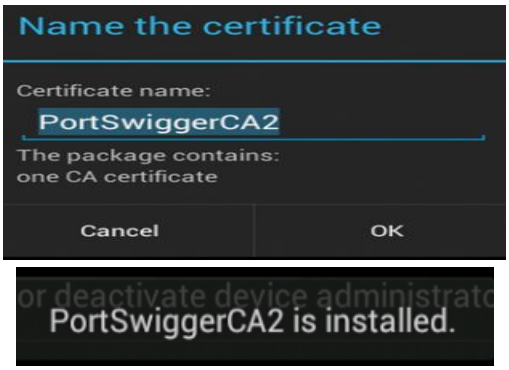


Figure 20 Installation du certificat

On nous demande alors de confirmer l'installation du certificat.



On confirme l'installation. Le certificat est alors copié de manière permanente dans le répertoire <cacerts-added>.

Une fois le device redémarré, la consultation du site visé ne provoque plus d'erreurs.

Figure 21 Validation du certificat

Concernant les versions d'Android inférieures à la 4, les manipulations sont plus complexes. Plus d'informations sont disponibles dans cet article : [EN] Using a custom certificate trust store on Android. Concernant l'installer d'un certificat sur un device réel, le lecteur intéressé pourra se référer aux articles suivants : Defeating SSL Certificate Validation for Android Applications / Intercepting and decrypting SSL communications between Android phone and 3rd party server / Mitmproxy avec Android.

2.5. L'analyse de l'application

2.5.1. L'analyse statique

L'analyse statique consiste à décompiler l'application afin d'en étudier le code. Cette méthode appelée reverse-engineering, a pour défaut que le temps passé à analyser le code peut être considéré comme du temps en moins pour tester la sécurité de l'application.

Normalement toutes les vulnérabilités du côté client peuvent être détectées sans avoir à lancer le code, mais dans la pratique, cela se révèle plus compliqué. L'objectif de l'analyse de code sera donc plutôt de détecter des problèmes de sécurité plus ou moins évidents et de se faire une idée sur la sécurité globale de l'application.

Pour ce type d'analyse, Il est recommandé d'utiliser le framework Androguard[12] qui permet de faciliter le travail d'analyse (permissions, instructions dangereuses, similarité entre 2 applications, etc.). Il existe aussi ARE[13], une machine virtuelle contenant une ancienne version d'Androguard.

Pour compiler/décompiler une application Android, le lecteur pourra se référer à l'annexe 2.

2.5.1.1. Outils d'analyse statique

voici quelques outils pour aider à l'analyse statique :

- Findbugs : Permet de trouver des bugs dans le code d'un programme Java.
- Android Lint : Permet de scanner un projet Android afin d'y trouver des bugs potentiels.
- Intellij Idea : Editeur Java.
- PMD : Analyseur de code-source

2.5.1.2. L'analyse du fichier Manifest.xml

Le fichier **AndroidManifest.xml** est un des fichiers les plus importants d'une application.
Ce fichier décrit :

- Les composants de l'application tels que les activités, les services, les mécanismes de communication, les communications entre composants, etc.
- Les processus fonctionnant avec les composants de l'application
- Les permissions requises par l'application pour interagir avec les composants du système
- Le niveau de l'API Android requis par l'application
- La liste des bibliothèques utilisées par l'application
- L'identification unique du package

Chaque application doit posséder un fichier **manifest** dans son répertoire système. Ce fichier présente les informations essentielles contenant l'application. L'analyse de ce fichier joue un rôle très important dans l'évaluation de la sécurité d'une application Android. Son analyse permet d'établir une liste des points d'entrée de composants à analyser. Par exemple :

[12] http://androguard.blogspot.fr/
[13] https://redmine.honeynet.org/projects/are

- L'application analysée envoie-t-elle des données sensibles à une application malicieuse ?
  - Analyser les intents qui contiennent des informations sensibles via les méthodes <sendBroadcast()>, <sendOrderedBroadcast()>, <sendStickyBroadcast()>, <sendStickyOrderedBroadcast()>.
  - Analyser l'exposition des données sensibles à une application malicieuse par le démarrage d'une activité (startActivity()) ou startService()…
- Les informations sensibles manipulées par l'application testée peuvent-elles être obtenue par une application malicieuse ?
  - Analyser le fournisseur de contenu exporté et non protégé par une permission. Une application malicieuse pourrait soit interroger directement le fournisseur de contenu, soit attendre la diffusion d'un intent vers les données du fournisseur
- Les données de l'application testée peuvent-elles être altérées par une application malicieuse ou lui faire réaliser des actions sensibles ?
  - Exemple : **Une faille détectée** permet à des cybercriminels jusqu'à effacer à distance les données du téléphone en invitant les utilisateurs à pointer sur une adresse URL, directement ou à travers un message textuel ou encore un QR Code. [14]

Pour rendre le fichier AndroidManifest.xml lisible, on peut utiliser l'outil AXMLPrinter :

```
C:\Tools Android>java -jar AXMLPrinter2.jar C:\Analyse\[illegible]\AndroidManif
est.xml > AndroidManifest.txt
```

AndroidManifest.txt - Bloc-notes

Fichier Edition Format Affichage ?

```
<?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.droid"
        >
        <uses-sdk
                android:minSdkVersion="13"
                android:maxSdkVersion="17"
                >
        </uses-sdk>
        <uses-permission
                android:name="android.permission.READ_PHONE_STATE"
                >
        </uses-permission>
```

Figure 22 Fichier AndroidManifest en clair

Voici quelques éléments importants à garder à l'esprit lors de l'analyse[15] :

Paramètre	Ce qu'il faut vérifier	Recommandations
Android : installLocation	Si la valeur est à « auto », cela signifie que l'application peut être installée sur un support externe, mais le système installera l'application sur un support interne par défaut. Si le support de stockage interne est plein, le système installera l'application sur	Utiliser le paramètre « interne » uniquement (internalonly)

[14] http://android.smartphonefrance.info/actu.asp?ID=2719
[15] http://resources.infosecinstitute.com/inside-android-applications/

	le support externe.	
android : protectionlevel	Indique le degré de risque du composant lié à cette permission. « dangerous » indique si le composant a accès à des données sensibles ou peut impacter de manière négative le fonctionnement du périphérique.[16]	Vérifier si la valeur est égale à « normale » ou « dangereuse ». Si égale à dangereuse, vérifier les permissions demandées.
android : persistent	Indique si l'application s'exécute tout le temps (true). La valeur par défaut est « faux »	Les applications ne doivent normalement pas avoir ce paramètre. Si c'est le cas, mettre à faux
android :<restoreAnyVersion >	Indique si une restauration des données sauvegardées est possible.	Mettre le paramètre à « vrai » permet au gestionnaire de sauvegarde de tenter une restauration des données si un conflit de version apparaît et que les données sont incompatibles.

2.5.1.3. Les vulnérabilités à rechercher

2.5.1.3.1. Mauvaise gestion des permissions

Une application qui demande des permissions abusives peut être considérée comme une mauvaise pratique de développement. Si celles-ci sont volontaires, on a affaire dans ce cas à une application malicieuse. Dans une application Android, les permissions requises sont explicitées dans le fichier <AndroidManifest.xml>. Du point de vue de la sécurité, ce fichier est intéressant car il définit les autorisations de l'application concernant d'autres applications ou celles protégées de l'API. C'est le rôle de l'application *Manifest explorer*.

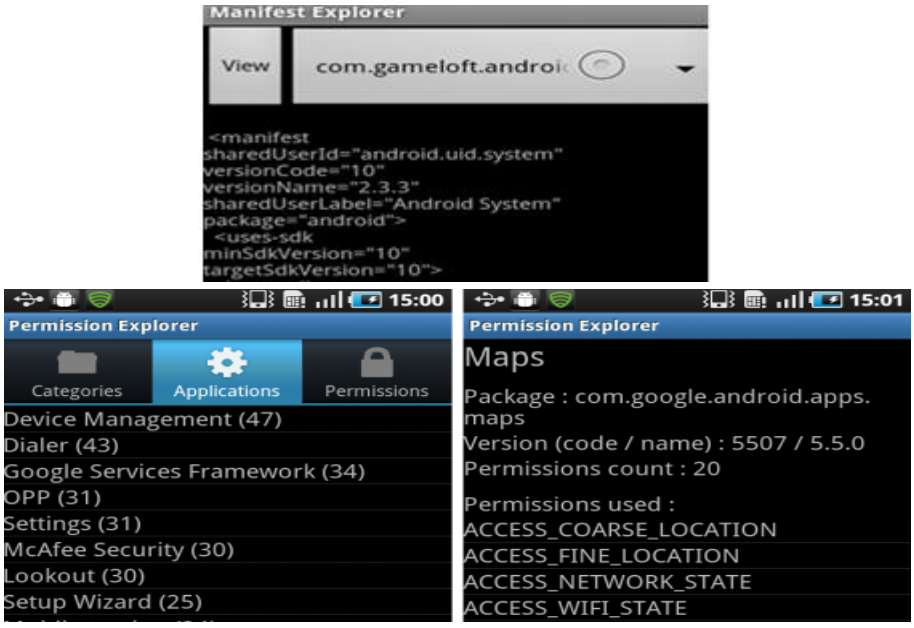


Figure 23 Application Manifest Explorer

[16] http://www.siteduzero.com/tutoriel-3-731717-les-permissions.html

La recherche des permissions peut donner de nombreux points d'entrées à un *pentester* lors d'un audit. Ci-dessous un exemple d'application qui a été rendue malicieuse.

Figure 24 A gauche, l'application originale. A droite, la même application infectée - Source : RSAConference.com

Parmi les permissions à risque sous Android, nous avons[17] :

- Accès au GPS : Voir la position géographique du terminal
- Appeler un n° (Call phone/privileged) : Initier un appel vocal
- Camera : Accès à la caméra
- Lire l'état du téléphone (Read phone state) : Lire le n° IMEI
- Installer/désinstaller des applications (Delete/install packages)
- Réinitialisation (Master clear) : Restaurer la configuration d'origine
- Enregistrement audio (Record audio) : Accéder au microphone
- Allumer / éteindre (Reboot / Shutdown) : Allumer ou éteindre le téléphone
- Envoi de sms (Send SMS) : Possibilité d'envoyer des messages
- Service qui coûte de l'argent

2.5.1.3.2. Données sensibles codées en dur

Il n'existe pas beaucoup d'outils mis à part Grep[18] pour aider l'auditeur dans cette phase. Parmi les données sensibles à rechercher dans le code-source, nous trouvons : clés de chiffrement, données utilisées lors du développement, mot de passe codés en dur, etc.

2.5.1.3.3. Mauvais stockage des fichiers

Une application Android peut utiliser plusieurs emplacements par défaut pour stocker ses données. Elle peut utiliser soit :

- **Le stockage externe (carte SD)**. Si l'écriture dans ce cas nécessite la permission <write_external_stockage>, la lecture n'en nécessite pas (les données stockées sur la carte SD sont accessibles à tout le monde).
- **Son répertoire personnel** (/data/data/<nom_du_package>)

Le stockage sur la carte SD n'étant pas sécurisé, on commence tout d'abord par rechercher la chaine <sdcard> afin de voir si ce chemin est utilisé dans l'application. Il faut aussi rechercher la chaine <getExternal> afin de prendre en compte les autres méthodes telles que <getExternalStorageDirectory()>,<getExternalfilesDir()>, <getExternalStoragePublicDirectory()>, etc. Si cette recherche donne des résultats, cela

[17] http://365.rsaconference.com/servlet/JiveServlet/previewBody/3472-102-1-4571/HT2-303.pdf

[18] http://pwet.fr/man/linux/commandes/grep

indique que l'application utilise probablement le stockage externe. Une analyse afin de déterminer quel est le type de données stockées est alors nécessaire.

Concernant le répertoire personnel de l'application, celui-ci ne peut par défaut être accédé que par l'application elle-même. Néanmoins Android permet grâce aux flags <mode_world_readable> et <mode_world_writable > de créer des données accessibles en lecture/écriture à d'autres applications. Ces flags sont donc à rechercher.

L'analyse des appels à <openFileOutputStream()>, <getSharedPreferences()>, <openOrCreateDatabase()>, permet de comprendre le fonctionnement et le moment où l'application écrit des fichiers.

NB : Sous Android 4.0, la permission <READ_EXTERNAL_STORAGE> est censée protéger l'accès en lecture au stockage externe.

Synthèse :

- Rechercher les flags <sdcard> et <getExternal> (stockage externe)
- Rechercher les flags <mode_world_readable> et <mode_world_writable> (stockage de l'application)

2.5.2. L'analyse dynamique

Au contraire de l'analyse statique et de l'étude du code-source de l'application, l'analyse dynamique consiste à étudier le comportement de l'application : appel de fonctions, chaines stockées en mémoire, trafic généré, etc. On utilise l'analyse dynamique pour l'étude de malwares (dans ce cas là, le recours à un environnement sécurisé de type *sandbox*[19] permet de contrôler les actions réalisées), et quand il n'est pas possible d'accéder au code-source de l'application (légalité du reverse engineering).

2.5.2.1. Outils d'analyse dynamique

Pour « culture », je rajoute ici un certain nombre d'outils permettant de réaliser une analyse dynamique (monitoring des actions réalisées) d'une application Android. On utilise ceux-ci pour en général tester des applications malveillantes dans un environnement protégé..

- Droidbox: Outil de type *Sandbox* pour les applications Android. Permet l'analyse dynamique (monitoring d'API, détection des fuites de données, analyse préliminaire statique, etc.)
- Mobile Sandbox : *Sandbox* pour applications mobile disponible en ligne.
- AndroidAuditTools: Outils pour analyse dynamique d'applications Android.

[19] http://fr.wikipedia.org/wiki/Sandbox_(s%C3%A9curit%C3%A9_informatique)

Figure 25 Analyse avec Mobile Sandbox

2.5.2.2. Accéder au menu caché de débogage

Il existe un menu « caché » afin d'accéder à de nombreuses options de débogage, ce qui peut se révéler utile pour le pentest de certaines applications (changement du user-agent[20], console Javascript, etc). On accède à ce menu par le navigateur et en entrant la commande:

Figure 26 Accéder au menu caché de debogage

Ensuite, en allant dans Menu, more, settings, on a accès à de nombreuses options :

Figure 27 Menu de debogage

Il peut être parfois utile de changer l'agent du navigateur (UAString) afin de simuler un appareil différent (ex : iPhone, iPad ou autre)

Figure 28 Simuler un autre périphérique par le changement du user-agent

[20] http://fr.wikipedia.org/wiki/User-Agent

2.5.2.3. Les vulnérabilités à rechercher

2.5.2.3.1. Communications réseau non sécurisées

On analyse tout d'abord les communications entre l'application et son/ses serveur(s). Pour cela, on peut utiliser les outils *tcpdump*[21] ou *wireshark*[22]. Si des données sensibles transitent, celles-ci doivent bien sûr être chiffrées.

Dans l'exemple suivant, la capture du trafic montre que des données sensibles sont envoyées sans que celles-ci ne soient chiffrées. Ces données sont issues d'une application d'espionnage que l'on installe sur un téléphone Android.

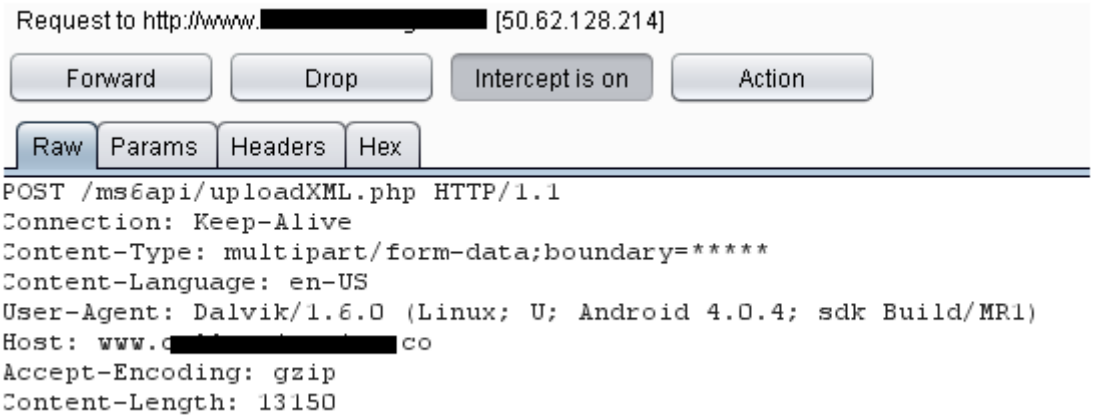
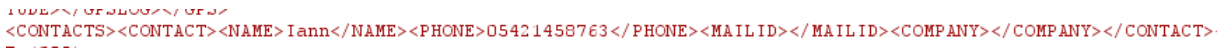


Figure 29 Capture du trafic avec Burp



On constate que les données (contacts dans l'exemple) transitent en clair.

Si les données sont bien chiffrées, il faut alors vérifier la validité du certificat utilisé. Si celui-ci n'est pas valide, cela ouvre des possibilités de Man In The Middle. En cas d'analyse du trafic avec un proxy du type *Burp*[23], il faut tout d'abord capturer le trafic sans ajouter de CA au TrustStore d'Android, cela permet de vérifier que la validité du certificat est bien vérifiée en cas d'erreur.

2.5.2.3.2. Présence d'informations sensibles dans les fichiers logs

Il peut arriver qu'il y ait des problèmes lors du passage de la phase de développement à la phase de production. Ces problèmes peuvent se traduire par la présence d'informations sensibles (mots de passe, token d'authentification, données métiers, etc.) La présence de ces données dans les logs d'Android peut être d'autant plus dangereuse si une application dispose de la permission <READ-LOGS>.

Par l'utilisation de la classe Log ou en utilisant la sortie System.out.printLn(), une application peut ajouter des entrées dans les logs du système.

[21] http://www.tcpdump.org/
[22] http://www.wireshark.org/
[23] http://www.portswigger.net/burp/

2.5.2.3.3. Failles web (injections SQL, Cross-Scripting)

Les applications Android ne sont pas exemptes des vulnérabilités classiques liées aux applications... Injections SQL et XSS peuvent être de la partie. Pou ceux qui souhaitent creuser le sujet :

- Some SQL injection in Android
- Query String Injection : Android Provider
- SQL injection in Java Application
- A Local Cross-Site Scripting Attack against Android Phones
- Android Gmail App : Stealing Emails via XSS

2.5.2.3.4. Fuzzing de l'application

Le fuzzing est une technique utilisée afin de tester le fonctionnement d'un programme en injectant des données aléatoires dans les entrées d'un programme. Elle permet de détecter les défauts d'un programme (crash ou génération d'erreurs).

Sur Android, l'application Intent Fuzzer permet de tester les activités (une ou toutes) d'une application.

Figure 30 Intent Fuzzer

2.5.2.3.5. IPC non sécurisées

Sous Android, les applications communiquent entre elles avec le mécanisme d'*intent*. Un *intent* est un message qui permet de :

- Transférer une donnée d'un composant à l'autre
- Demander une donnée à un composant
- Demander à un composant de réaliser une action sur une donnée

Dans la capture suivante, un clic sur le bouton « google » lance une recherche google avec la chaine de caractère passée en paramètre (*intent*)[24].

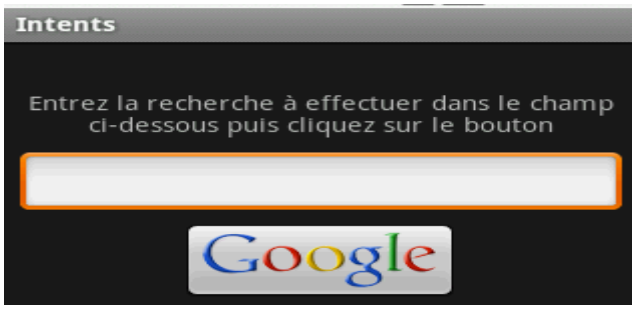


Figure 36 Exemple d'intent

[24] http://www.pointgphone.com/tutoriel-android-introduction-intents-7779

Ce mécanisme rend la plate-forme Android plus flexible, mais élargit aussi la surface d'attaque d'une application.

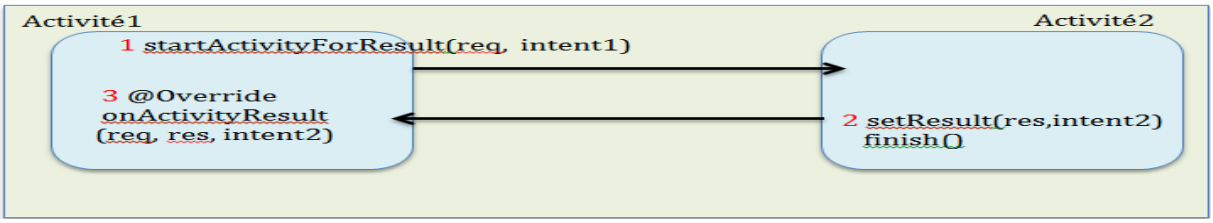


Figure 37 Mécanisme d'intent sous Android

Si vous désirez creuser le sujet, vous pouvez consulter l'article suivant : [FR] Activite Intent BroadcastReceiver Fragments.

Concernant les risques liés aux *intents*, on commence tout d'abord par analyser le fichier AndroidManifest.xml, car c'est dans ce fichier que sont définies les permissions de l'application (permissions demandées, définies, et différents composants). Parmi les 2 attributs les plus intéressants pour un auditeur, se trouvent <android :exported> (définit si le composant est accessible à d'autres applications) et <android :permission> (protège le composant par une permission). Ci-dessous un extrait d'un fichier AndroidManifest issu d'un malware[25] :

```
<receiver android:name="com.google.android.smart.LcbiveReceivecr">
    <intent-filter>
        <action android:name="android.intent.action.CFF" />
        <action android:name="android.intent.action.PHONE_STATE" />
        <action android:name="android.intent.action.SIG_STR" />
        <action android:name="android.intent.action.SERVICE_STATE" />
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
        <action android:name="android.intent.action.BATTERY_CHANGED" />
        <action android:name="android.intent.action.DATE_CHANGED" />
        <action android:name="android.intent.action.NEW_OUTGOING_CALL" />
        <action android:name="android.intent.action.REBOOT" />
        <action android:name="android.intent.action.TIME_CHANGED" />
        <action android:name="android.intent.action.WALLPAPER_CHANGED" />
    </intent-filter>
</receiver>
```

Pour aider à l'analyse des vulnérabilités basées sur les communications inter-composant (*intent*), il existe l'outil d'analyse statique COMDROID. Cet outil permet aux développeurs de connaitre les intents vulnérables et de donner plus de détails :

Unauthorized Intent Receipt	
Intent type	Potential vulnerability
Sent Broadcasts	Broadcast Theft (without data) Broadcast Theft (with data)
Sent Activity requests	Activity Hijacking (without data) Activity Hijacking (with data)
Sent Service requests	Service Hijacking (without data) Service Hijacking (with data)
Intent Spoofing	
Component type	Potential vulnerability
Exported Broadcast Receivers	Broadcast Injection (without data) Broadcast Injection (with data) System Broadcast without Action Check
Exported Activities	Activity Launch (without data) Activity Launch (with data)
Exported Services	Service Launch (without data) Service Launch (with data)

Figure 31 Liste des différentes vulnérabilités associée à chaque type d'intent et à son composant

[25] http://resources.infosecinstitute.com/rootsmart-android-malware/

Un papier de recherche est d'ailleurs consacré à ce thème : [EN] Analyzing Inter-Application Communication in Android.

Concernant les composants fournisseurs de données (provider), ceux-ci sont par défaut exportés. Par contre, certains composants tels que les récepteurs de broadcast peuvent être déclarés dynamiquement dans l'application (ce qui implique qu'ils n'apparaitront pas dans le fichier manifest). Ci-dessous un extrait de l'analyse d'une application malveillante avec une sandbox online. Le rapport est disponible à cette adresse.

Used Intents:

android.intent.action.BOOT_COMPLETED
android.intent.action.NEW_OUTGOING_CALL
android.intent.action.SCREEN_ON
android.intent.action.SCREEN_OFF
android.intent.action.PHONE_STATE
android.intent.action.PACKAGE_ADDED
android.intent.action.PACKAGE_REPLACED
android.intent.action.PACKAGE_REMOVED
android.intent.action.MAIN
android.intent.category.LAUNCHER

Figure 39 Intents utilisés par une application malveillante

Un composant peut aussi vérifier dynamiquement par la méthode <checkCallingPermission> qu'une application qui l'appelle dispose bien d'une permission donnée.

3. Conclusion

Le pentest d'une application Android prend 2 formes bien distinctes : l'analyse de l'application elle-même, et l'analyse du trafic client-serveur (backend) utilisé par l'application. Concernant l'analyse du code-source de l'application, on étudiera si les bonnes pratiques de développement ont bien respectées (le lecteur intéressé pourra se référer au document suivant : Menaces et failles du système Android). Une fois l'interception et l'analyse du trafic chiffré réalisé, on se retrouve alors dans le cas d'un audit web classique (recherche d'injections, management des sessions, test de l'authentification, etc.).

Il est d'ailleurs intéressant de noter que les sites mobiles sont en général moins bien sécurisés que les sites dits "normaux" (retour d'expérience de mission).

A. Annexes

A.1. Emplacement de données sous Android

Liste issue de l'article : Android Forensics

```
# find data -name "*.db" -print /data/data/
data/data/com.google.android.browser/app_appcache/ApplicationCache.db
data/data/com.google.android.browser/app_databases/Databases.db
data/data/com.google.android.browser/app_geolocation/CachedGeoposition.db
data/data/com.android.providers.calendar/databases/calendar.db
data/data/com.android.providers.contacts/databases/profile.db
data/data/com.android.providers.contacts/databases/contacts2.db
data/data/com.android.providers.downloads/databases/downloads.db
data/data/com.google.android.email/databases/EmailProvider.db
data/data/com.google.android.gm/databases/internal.mometan@gmail.com.db
data/data/com.google.android.gm/databases/mailstore.mometan@gmail.com.db
data/data/com.google.android.gm/databases/webviewCookiesChromium.db
data/data/com.google.android.music/databases/music.db
data/data/com.android.providers.telephony/databases/telephony.db
data/data/com.android.providers.telephony/databases/mmssms.db
data/system/accounts.db
```

Figure 32 Emplacements des informations sous Android

A.2. Compilation/décompilation d'une application

A.2.1. Compiler une application

Il peut être toujours utile de savoir comment compiler une application Android quand on dispose des sources. Pour compiler une application Android (format APK) à partir de son code source, on utilise l'environnement de développement Eclipse[26]. On crée tout d'abord un nouveau projet Android dans lequel on indique que les sources sont déjà existantes.

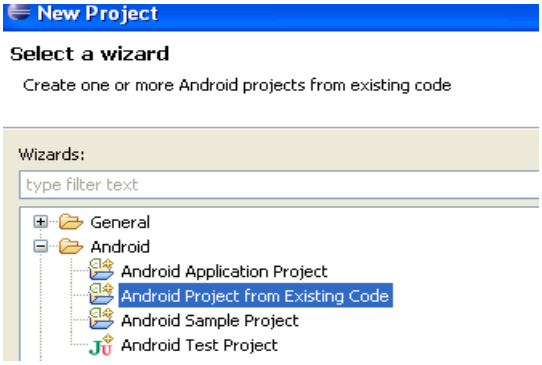
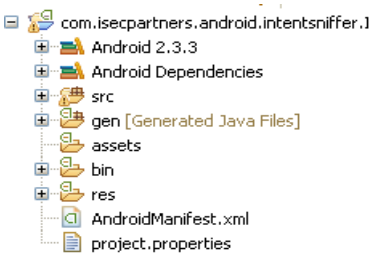


Figure 33 Compilation d'une application Android

[26] http://www.tutomobile.fr/installer-le-sdk-android-sur-eclipse-tutoriel-android-n%C2%B01/09/06/2010/

Une fois le code de l'application chargé, on va dans le menu « File », puis Export : Android -> Export Android Application.



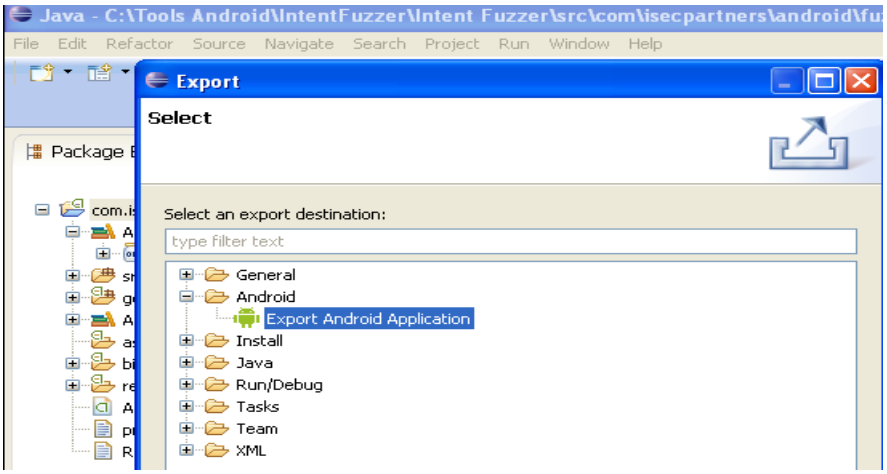


Figure 34 Compilation d'une application Android

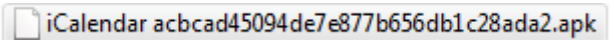
On obtient alors un fichier apk que l'on peut installer sur un device Android.

```
C:\Program Files\Android\android-sdk\platform-tools>adb install "c:\Tools Andro
d\com.isecpartners.android.fuzzers.IntentFuzzer.apk"
78 KB/s (17677 bytes in 0.218s)
        pkg: /data/local/tmp/com.isecpartners.android.fuzzers.IntentFuzzer.apk
Success
```

Figure 35 Installation de l'application générée

A.2.2. Décompiler une application

Comme dit précédemment, les applications Android sont sous forme d'un fichier APK[27].



Un fichier APK est un fichier ZIP basé sur le format JAR. Il contient en général les fichiers suivants :

- META-INF (dossier)
  - MANIFEST.MF : le fichier manifeste (?)
  - Cert.rsa : Le certificat de l'application
  - Cert.sf : la liste des ressources
- Res (répertoire contenant les ressources non compilées
  - AndroidManifest.xml : Fichier manifeste additionnel (décrit le nom, la version, les droits d'accès, etc.)
  - Classes.dex : le fichier class compilé dans le format dex
  - Resources.arsc : Fichier précompilé des ressources

[27] http://en.wikipedia.org/wiki/APK_%28file_format%29

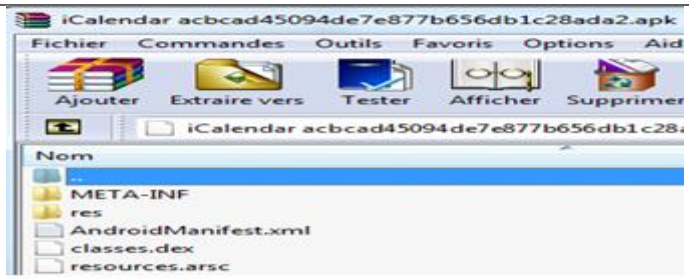


Figure 36 Contenu de l'application iCalendar

La première étape est de décompresser l'archive APK, pour cela un utilitaire de type winzip, winrar, etc. est largement suffisant.. Une fois décompilée, on obtient un fichier classes.dex qu'il va falloir décompiler.

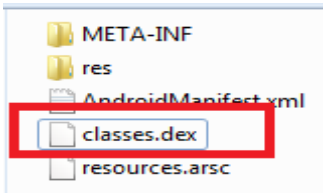


Figure 37 Archive APK décompilée

Ce fichier est le résultat de la compilation du code-source[28]. Pour le décompiler, nous utilisons le décompilateur Java Dex2jar.

```
D:\Projets\TP Android\Malwares\Tools\dex2jar-0.0.9.8>dex2jar.bat samples\classes
.dex
dex2jar version: translator-0.0.9.8
dex2jar samples\classes.dex -> samples\classes_dex2jar.jar
Done.
```

Figure 38 Décompilation de l'application Android

Une fois l'application décompilée, nous pouvons utiliser l'outil JD-GUI qui va nous permettre de naviguer dans le code-source.

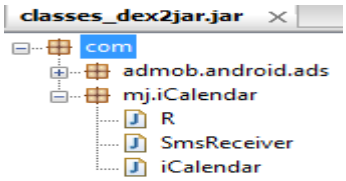


Figure 39 Application décompilée

Il est aussi possible d'utiliser l'outil Eclipse afin de naviguer dans le code source.

Un autre outil très utile pour décompiler une application Android est apktool. Contrairement à Dex2jar, apktool rend accessible l'ensemble de l'application (fichier manifeste et code-source)

```
C:\Tools Android\apktool>apktool d c:\Samples\OmeGaSpy\OmegaTarget4x.apk
I: Baksmaling...
I: Loading resource table...
I: Loaded.
I: Loading resource table from file: C:\Documents and Settings\Administrateur\ap
ktool\framework\1.apk
I: Loaded.
I: Decoding file-resources...
I: Decoding values*/* XMLs...
I: Done.
I: Copying assets and libs...
Appuyez sur une touche pour continuer...
```

Figure 40 Décompilation d'une application avec apktool

[28] http://www.pythagore-fd.fr/documents/extraits/pdf/formation-UX128-LinuxMobile_Android-dex_native.pdf