Sécurité du système Android

Nicolas Ruff

EADS Innovation Works
nicolas.ruff(@)eads.net

1 Introduction

Le système Android connait un succès croissant auprès des développeurs d'applications embarquées (de type ordiphones, mais également tablettes, téléviseurs connectés, et pourquoi pas demain réfrigérateurs, etc.).

Ce succès commercial s'accompagne d'un nombre croissant de publications autour de la sécurité du produit ... ainsi que d'un nombre croissant d'incidents de sécurité. En 2010, on peut citer par exemple la publication de l'application PhoneCreeper ou du « bot » Geinimi ... ainsi que le challenge SSTIC bien sûr.

Cet article se propose de réaliser un tour d'horizon des attaques connues contre le système Android, ainsi que des faiblesses intrinsèques héritées de la structure du marché de la téléphonie mobile. Il propose également une méthodologie d'audit sécurité des applications Android.

Bien que les choix technologiques initiaux puissent difficilement être remis en question, il faut toutefois noter que le système Android est en perpétuelle évolution dans un marché où les cycles de vie sont l'ordre de 6 mois. Le lecteur est donc invité à se tenir à jour des évolutions postérieures à la rédaction de cet article.

Note : sauf mention contraire, tous les tests ont été réalisés avec les plateformes suivantes :

- Emulateur officiel fourni avec le SDK
- Smartphone LG GW620 sous Android 1.5
- Smartphone HTC Desire Z sous Android 2.2

Chaque équipement physique étant « personnalisé » par les constructeurs (par rapport aux sources « officielles » fournies par Google), votre expérience utilisateur peut différer de la mienne.

2 Quelques mots d'histoire

Le système Android n'a pas été développé par Google mais par Android, Inc. - une société californienne créée en 2003 par des intervenants expérimentés dans le domaine

des télécoms. La société Android, Inc. a été rachetée par Google en 2005, mais la sortie officielle du « produit » Android n'a eu lieu qu'en 2007 - en même temps que la création de l'Open Handset Alliance [1], chargée de faire « vivre » ce produit.

Au travers de cet historique, on peut comprendre que les choix de conception initiaux (et donc structurants) n'ont pas été faits par Google.

3 Principes d'implémentation

3.1 Architecture ARM

Le système Android est disponible pour plusieurs architectures CPU (comme x86, ARM et MIPS). Toutefois l'écrasante majorité des équipements en circulation équipés du système Android (à savoir les téléphones portables) disposent d'un processeur ARM. Cette tendance gagne d'ailleurs les NetBooks (appelés « SmartBooks » dans ce cas). A terme, la domination insolente de l'architecture x86/x64 est donc menacée - Microsoft annonce d'ailleurs qu'il y aura une version ARM de Windows 8 (mais ceci est une autre histoire). Dans la suite de cet article, seule l'architecture ARM sera donc étudiée.

L'architecture ARM est un *design* de la société ARM Ltd, qui ne fabrique pas elle-même ses processeurs mais les revend sous licence. De nombreux constructeurs proposent donc des processeurs ARM implémentant tout ou partie de l'une des spécifications disponibles - actuellement ARMv5, ARMv6 ou ARMv7 pour le cœur - avec différentes options telles que la présence d'un FPU, d'un DSP, d'un support natif du bytecode Java (technologie Jazelle), etc.

La nomenclature des processeurs ARM est déroutante de prime abord [2]. Ainsi le processeur ARM7-TDMI, qui a rencontré un grand succès commercial (il équipe l'iPod et la Nintendo DS par exemple), est construit sur un cœur ARMv4 (comme son nom ne l'indique pas).

Au final, le parc installé est donc très hétérogène et le développeur d'applications (ou d'attaques) ne peut s'appuyer que sur le plus petit dénominateur commun.

En ce qui concerne strictement la sécurité du processeur ARM, notons les points suivants :

– Les caches d'instructions et de données sont généralement séparés dans les implémentations ARM courantes. Ceci rend plus compliqué l'écriture de code auto-modifiant (de type shellcode encodé par un XOR) ;

1. http://www.openhandsetalliance.com/
2. http://en.wikipedia.org/wiki/ARM_architecture

- L'encodage des instructions est de taille fixe (16 ou 32 bits, selon le mode du processeur : THUMB ou ARM). Il est possible de « jouer » avec chaque bit à l'intérieur d'une instruction (par exemple pour la rendre conditionnelle), ce qui facilite l'écriture de shellcodes sans octets nuls ;
- La pile peut être ascendante ou descendante. Dans la pratique, toutes les implémentations ARM dont j'ai connaissance ont une pile qui fonctionne comme sur architecture x86 (donc descendante) ;
- Le passage de paramètres s'effectue par registres ;
- Enfin certaines implémentations du processeur ARM supportent le bit XN (eXecute Never), apparu avec ARMv6. Ce bit est équivalent au bit NX/XD sur architecture x86.

Tous ces points ne sont pas spécifiques au système Android et sont déjà largement documentés par ailleurs. La société iSec avait déjà publié un article sur l'écriture de shellcodes ARM en... 2001 [3] !

3.2 L'héritage GNU/Linux

Android est un système d'exploitation « Open Source » (licence Apache v2), basé sur le système GNU/Linux, avec lequel il partage d'ailleurs l'hébergement des sources [4] - même si la pérennité de ce modèle pourrait être remise en cause compte-tenu des divergences croissantes entre les développements réalisés par Google et la branche stable du noyau [5].

Les différences avec une distribution Linux traditionnelle sont toutefois assez significatives :

1. La librairie C n'est pas la GNU/LibC traditionnelle, mais une version développée par Google et dénommée « Bionic ».

   On peut supposer que les déboires de Debian avec la GNU/LibC (et particulièrement le support ARM [6]) ne sont pas totalement étrangers à ce choix. La licence GNU pose également un problème potentiel, c'est pourquoi Google s'est plutôt inspiré du code issu des projets *BSD (se reporter au répertoire bionic/libc dans les sources d'Android pour plus de détails).

2. Les applications Java tierce partie doivent nécessairement être compilées en bytecode Dalvik.

3. http://www.isec.pl/papers/into_my_arms_dsls.pdf
4. http://android.git.kernel.org/
5. http://www.kroah.com/log/linux/android-kernel-problems.html
6. http://linux.slashdot.org/article.pl?sid=09/05/06/2050216

Ce bytecode s'exécute dans une JVM spécifique à Android, dénommée « Dalvik » en référence à un village islandais dont est issue la famille du chef de projet. Il s'agit d'une machine virtuelle à registres et non à pile comme la JVM de Sun. Le bytecode Dalvik est par ailleurs très similaire au bytecode Java... mais pas complètement identique non plus.

Le choix d'une JVM comme cible de développement s'explique par plusieurs raisons parfaitement acceptables : facilité de développement de nouvelles applications, abstraction de la plateforme matérielle (les variantes et les révisions du processeur ARM étant nombreuses), possibilité de vérifier des propriétés de sécurité sur le bytecode, etc.

Toutefois il faut noter que depuis le rachat de la technologie Java par Oracle, la bataille juridique fait rage entre les deux éditeurs [7].

Le développement de code natif est également possible au travers du NDK. Il n'est toutefois pas possible de distribuer sur la MarketPlace des applications 100% natives : le code natif doit utiliser les API JNI pour s'interfacer avec une application Java « hôte ».

3. Toutes les applications doivent être signées numériquement.

   Le mécanisme de signature Android n'est pas basé sur une PKI mondiale qui serait difficile à maintenir et commercialement suicidaire. Un certificat autosigné généré par le développeur est acceptable, comme nous le verrons plus loin en abordant le modèle de sécurité du système.

4. Un équipement « physique » n'est pas entièrement « Open Source ».

   Le milieu du matériel et des couches basses (ex. pile de protocoles GSM) est très concurrentiel et reste traditionnellement assez hostile au concept de « l'Open Source ». Un smartphone vendu par un constructeur tiers va embarquer du code propriétaire (essentiellement des pilotes mais aussi parfois des interfaces graphiques).

   Même les téléphones de développement commercialisés par Google (série des Nexus) perdent une partie de leurs fonctionnalités lorsqu'ils sont recompilés uniquement à partir des sources publiques.

   La situation n'est pas très différente de celle d'un PC moderne, dans lequel de nombreux composants embarquent des firmwares propriétaires. On peut noter toutefois que ces composants ont fait l'objet de nombreuses recherches ces dernières années. Des risques de sécurité avérés ont été exhibés dans des composants tels que Intel vPro [5] , les contrôleurs claviers [2,3] , les cartes réseau [4,6] , etc.

7. http://www.computerworld.com/s/article/9180678/Update_Oracle_sues_Google_over_Java_use_in_Android

Plus proche du téléphone, la compromission du baseband a été démontrée possible récemment [7] . La sécurité de la plateforme ne peut pas s'envisager sans la sécurité des composants matériels qui la compose.

5. Les interfaces avec le système d'exploitation ne sont pas celles de Linux.
   On peut même raisonnablement affirmer que l'API Java exposée aux applications est radicalement différente de celle utilisée dans la programmation Unix traditionnelle. L'interface graphique n'est pas X11 (heureusement :). Les communications inter-processus doivent passer par un mécanisme spécifique appelé Binder. De nombreux concepts d'exécution adaptés aux contraintes d'un smartphone ont été modélisés (ex. processus de type Activity, Service, Broadcast Receiver ou Content Provider).

3.3 Développement d'applications

Le développement sous Android s'effectue majoritairement en langage Java, ce qui contribue à la popularité de la plateforme auprès des développeurs. Un SDK est fourni gratuitement par Google pour les trois systèmes d'exploitation majeurs (Windows, Mac OS X et Linux). Ce SDK peut s'intégrer dans l'IDE bien connu Eclipse, ou fonctionner en ligne de commande. Il dispose d'un émulateur (basé sur QEmu) ainsi que d'outils de mise au point puissants (comme DDMS).

Après compilation, l'utilitaire DX (fourni dans le SDK) permet de convertir le bytecode Java en bytecode Dalvik. L'opération inverse n'est pas possible avec les outils du SDK officiel.

L'application installable sur le téléphone est empaquetée dans un fichier APK, qui inclut le bytecode de l'application elle-même (fichier CLASSES.DEX), le manifeste d'application, ses ressources, et les signatures numériques. Le concept est identique à celui des fichiers JAR – les fichiers APK sont d'ailleurs eux-mêmes des archives ZIP.

Le développement d'applications est un peu déroutant de prime abord - l'application ne disposant pas d'un point d'entrée fixe (de type `main()`) mais s'enregistrant plutôt auprès du système pour répondre à des évènements appelés Intents dans la documentation officielle.

Il est également possible d'intégrer du code natif dans une application Java par une interface JNI et le kit de développement officiel appelé NDK (Native Development Kit). De nombreuses applications cryptographiques exploitent cette opportunité pour améliorer les performances des algorithmes « gourmands » et assurer la sécurité des éléments cryptographiques en mémoire (le mécanisme de *garbage collector* de Java étant difficilement compatible avec l'effacement sécurisé de données en mémoire).

Il n'est pas possible de développer une application 100% native et de la mettre à disposition sur une MarketPlace. Il reste assez simple de compiler une application

100% native, mais un code Open Source existant pour Linux ne va pas nécessairement recompiler et fonctionner sans modification sur une plateforme Android. L'absence de `/bin/sh` et de `/etc/passwd` sont par exemple deux écueils auxquels j'ai été confrontés.

L'objectif n'est pas ici d'écrire un tutoriel pour le développement d'applications Android, d'autant que la documentation Google est très bien faite [8].

3.4 Modèle de sécurité

Le modèle de sécurité Android n'a guère évolué depuis ses origines, il a donc déjà été abondamment commenté [1]. Son étude reste néanmoins un préalable obligatoire à toute analyse de risques sur cette plateforme.

Signature numérique Les applications sont signées numériquement. Ceci est source de confusion dans l'esprit du grand public et même des informaticiens, qui associent généralement la présence d'un certificat à celle d'une sécurité inviolable, probablement à cause de la propagande réalisée par les acteurs du e-commerce autour du protocole HTTPS (les fameuses vertus du cadenas jaune, et plus récemment de la barre d'adresse verte).

En pratique, tous les systèmes de signature numérique apparus récemment dans les marchés grand public (ex. programme « Symbian Signed », signature des applications iPhone, signature des pilotes Windows 64 bits) n'offrent aucune sécurité a priori. Le coût d'acquisition d'un certificat est marginal, et les vérifications opérées quasiment nulles. Le certificat sert essentiellement d'identifiant unique pour permettre la révocation a posteriori.

Afin d'éviter les problèmes d'expiration et de renouvellement des clés, Google impose même l'utilisation de certificats de signature dont la date d'expiration dépasse le 22 octobre 2033 [9].

Cloisonnement A l'installation, chaque application se voit attribuer un compte Unix (`uid`). L'isolation entre applications est rendue possible par les mécanismes de sécurité natifs du système Unix.

L'ensemble des applications signées par le même certificat s'exécutent sous la même identité de groupe Unix (`gid`). Les interactions possibles entre applications signées par le même certificat sont nombreuses [10] – elles peuvent ainsi partager le même `uid` [11], ce qui ouvre la voie à la création d'applications malveillantes k-aires : chaque

8. http://developer.android.com/
9. http://developer.android.com/guide/publishing/app-signing.html
10. http://developer.android.com/guide/topics/security/security.html
11. http://developer.android.com/reference/android/R.attr.html#sharedUserId

application ne possédant qu'un jeu limité de permissions (jugées peu dangereuses individuellement), mais la combinaison de toutes ces applications dans le même processus formant une « super application » pouvant par exemple exfiltrer toutes les données du téléphone.

Révocation Compte-tenu de l'accroissement du nombre d'applications malveillantes sur la MarketPlace officielle, Google fait un usage de plus en plus fréquent du *kill switch*[12] – fonction qui permet d'éliminer à distance toutes les instances d'une application identifiée par son certificat.

Pour ceux que cette fonction intrigue, je signale que le mécanisme sous-jacent a déjà été décortiqué en détail [18,19] . Le cœur du mécanisme repose sur le processus GTalkService, qui reçoit et traite les messages `REMOVE_ASSET` (et `INSTALL_ASSET`). La conclusion en est que toute personne en position d'effectuer un *man-in-the-middle* SSL avec un certificat valide peut émettre de tels messages. La liste des autorités de confiance du système se trouve dans le fichier `/system/etc/security/cacerts.bks` et ne peut pas facilement être modifiée[13]. Toutefois cette liste contient (sur mon téléphone) 58 autorités de nature variée (entreprises, gouvernements, etc.). De plus, une application malveillante ayant élevé ses privilèges vers `root` pourrait probablement bloquer l'utilisation du *kill switch* sur un équipement donné, une fois les mécanismes sous-jacents connus et documentés.

Par le passé, la sécurité basée sur la révocation a montré ses limites[14]. En effet cela suppose que l'équipement cible dispose d'une connexion à un réseau de données, et qu'il soit configuré pour en faire usage. Ce sont des hypothèses assez fortes.

3.5 Les permissions

Considérations générales Le modèle de sécurité des applications tierce partie est essentiellement déclaratif. Un fichier de « manifeste » décrit les permissions maximales requises par l'application. Google définit une centaine de permissions possibles par défaut[15], mais les constructeurs sont libres d'en ajouter également, ce qui peut engendrer des risques spécifiques à un modèle de téléphone donné.

La commande de base permettant de manipuler les permissions sur le téléphone s'appelle `pm`. Voici les options qu'elle offre :

```
$ adb shell pm
```

12. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html
13. http://www.mcbsys.com/techblog/2010/12/android-certificates/
14. http://www.f-secure.com/weblog/archives/00001918.html
15. http://developer.android.com/reference/android/Manifest.permission.html

```
usage: pm [list|path|install|uninstall]
       pm list packages [-f]
       pm list permission-groups
       pm list permissions [-g] [-f] [-d] [-u] [GROUP]
       pm list instrumentation [-f] [TARGET-PACKAGE]
       pm list features
       pm path PACKAGE
       pm install [-l] [-r] [-t] [-i INSTALLER_PACKAGE_NAME] [-s] [-f] PATH
       pm uninstall [-k] PACKAGE
       pm enable PACKAGE_OR_COMPONENT
       pm disable PACKAGE_OR_COMPONENT
       pm setInstallLocation [0/auto] [1/internal] [2/external]

(...)
```

A titre d'exemple, voici les permissions définies par HTC sur l'une de mes plateformes de test :

```
$ adb shell pm list permissions | grep htc

permission:com.htc.android.mail.permission.READ_MAIL
permission:com.htc.Manifest.permission.BROADCAST_KEYIN_EVENT
permission:com.htc.Manifest.permission.BROADCAST_MOTION_EVENT
permission:com.htc.Manifest.permission.BLOCK_KEYBOARD_EVENT
permission:com.htc.android.mail.permission.WRITE_ACCOUNT
permission:com.htc.android.mail.permission.READ_ACCOUNT
permission:com.htc.htctwitter.permission.useprovider
permission:com.htc.Manifest.permission.BROADCAST_TRACKBALL_EVENT
permission:com.htc.internal.permission.HTC_APP_PMEM
permission:com.htc.android.mail.permission.WRITE_MAIL
```

On notera également que cette plateforme contient des permissions a priori spécifiques à l'opérateur (américain) Sprint - alors qu'il s'agit d'un téléphone acheté en France, sans abonnement :

```
$ adb shell pm list permissions | grep sprint

permission:com.sprint.internal.permission.SLOT0
permission:com.sprint.internal.permission.PLATFORM
permission:com.sprint.internal.permission.SYSTEMPROPERTIES_WRITE
permission:com.sprint.internal.permission.SYSTEMPROPERTIES
permission:com.sprint.internal.permission.CONNECTIONMANAGER
permission:com.sprint.internal.permission.OMADM
permission:com.sprint.android.permission.DISABLE_HDMI
```

L'utilisateur est libre d'accepter ou de refuser l'application, mais il ne peut pas modifier la liste des permissions demandées (sauf à modifier l'application et à la re-signer avec sa propre clé comme nous le verrons plus tard).

On peut raisonnablement estimer que la plupart des utilisateurs ne sont pas en mesure d'effectuer un choix éclairé à ce stade.

Sécurité théorique L'une des questions qu'on peut se poser sur le système de permissions tel qu'il a été défini est l'existence (ou non) d'un modèle théorique sous-jacent cohérent [9,8]. Est-ce que chaque permission correspond bien à un ensemble de tâches uniques et strictement délimitées ? C'est la même question qui se pose avec le modèle des capabilities sous Linux, et dans ce deuxième cas la réponse est « non » : `CAP_SYS_MODULE` permet par exemple de charger des modules noyau arbitraires, et donc de compromettre entièrement l'intégrité du noyau.

Dans le cas d'Android, on peut suspecter que la permission `SYSTEMPROPERTIES_WRITE` définie par l'opérateur Sprint (vue précédemment) a probablement une intersection non nulle avec la permission `WRITE_SETTINGS` définie par Google, tout en étant probablement inconnue des outils d'analyse automatique d'innocuité...

L'intérêt pratique d'une étude mathématique sur les permissions Android est toutefois limité. En effet, les deux risques majeurs rencontrés dans la nature sont :

- Des applications malveillantes qui demandent des permissions excessivement larges, et abusent de la confiance des utilisateurs ;
- Des applications malveillantes sans aucune permission qui exploitent des failles système pour obtenir l'accès au compte `root` et contourner entièrement le modèle de sécurité.

Ces risques sont partagés avec tous les autres modèles de sécurité en usage dans l'industrie (SELinux sous Linux, UAC sous Windows, etc.) :

- Ces modèles se heurtent à l'ergonomie des interfaces et à la prise de décision éclairée par les utilisateurs finaux ;
- L'implémentation de ces modèles exige un effort de conception important, tandis que les attaquants les contournent entièrement en violant les hypothèses de conception (en particulier l'hypothèse que le noyau du système d'exploitation est exempt de bogues).

Autres limites du modèle Presque toutes les applications existantes exigent un accès illimité au réseau, très souvent pour récupérer du contenu publicitaire. En pratique cela signifie que le risque posé par une application ne se limite pas à un ensemble de permissions statiques : par exemple, un contenu malveillant pourra être envoyé à tout moment au moteur de rendu WebKit à travers le canal publicitaire, et provoquer ainsi la prise de contrôle d'un téléphone donné au travers une application a priori « saine ».

Les permissions demandées peuvent changer à l'occasion d'une mise à jour. Or la plupart des utilisateurs font confiance aux applications qu'ils ont déjà installées. Depuis la version 2.2 d'Android, l'utilisateur peut autoriser les applications à se mettre à jour automatiquement. Fort heureusement, la mise à jour automatique est

bloquée si les permissions demandées par l'application changent. De mon point de vue, le mécanisme de mise à jour automatique peut donc être considéré comme plutôt bénéfique pour l'écosystème.

Enfin le code natif attaché à une application (dans le cas d'utilisation du NDK) est essentiellement « non vérifiable », contrairement au bytecode Dalvik. Le modèle de sécurité Unix vient donc s'ajouter au modèle de permissions Android.

4 Panorama des risques

J'identifie trois risques de sécurité majeurs applicables aux ordiphones « modernes » :

1. La structuration du marché de la téléphonie.
2. Les failles logicielles.
3. Les applications tierce partie.

Nous allons passer en revue ces trois risques. Le troisième risque est plus spécifique à l'écosystème Android, dans lequel le rôle de chaque acteur est bien séparé : fournisseur de matériel, fournisseur de logiciel, fournisseur de service.

Il se vérifie moins dans des écosystèmes comme celui de l'iPhone et du BlackBerry, dans lesquels les fournisseurs du matériel et du logiciel sont les mêmes.

4.1 Marché de la téléphonie

Plusieurs acteurs majeurs interviennent dans le marché de la téléphonie mobile :

1. Les fabricants de matériels.
2. Les fabricants de logiciels (en l'occurrence Google).
3. Les circuits de distribution (pour l'instant, les opérateurs de téléphonie mobile).
4. Les développeurs d'applications.
5. Les fournisseurs de contenus et de services (régies publicitaires, services en ligne, etc.).

Nous allons passer en revue les intérêts de ces acteurs, et le rôle qu'ils jouent dans la sécurité de la plateforme, à l'exception des derniers qui sont trop divers et variés.

Il faut noter que si demain Android venait à équiper d'autres biens de consommation courante (ex. téléviseurs, réfrigérateurs, etc.), les parties prenantes (et donc les conclusions de cette étude) ne seraient pas fondamentalement différentes.

Fabricants de matériels Les fabricants de matériels vivent dans un monde extrêmement concurrentiel, et doivent minimiser le coût de développement de leurs produits, pour que l'opérateur puisse au final proposer son téléphone à 1 . Par ailleurs ils doivent également minimiser le « *time to market* » : un téléphone portable se démode très vite, la durée de vie commerciale d'un système est de l'ordre de 6 mois.

En conséquence logique, les développements sont bâclés. Les systèmes poussés en production sont à peine testés, et les fonctions de débogage restent souvent activées (comme en témoigne le shell `root` disponible sur le port TCP/12345 des séries HTC Evo et HTC Hero commercialisées par Sprint [16]).

En règle générale, aucun support technique n'est offert après la sortie d'un téléphone. Les mises à jour officielles de firmware sont rarissimes dans le domaine de la téléphonie mobile - à l'exception notable d'Apple, qui propose des mises à jour pour le système iOS dans la fenêtre de support de ses équipements, à savoir $2\frac{1}{2}$ ans. Ceci conduit à la prolifération de firmwares non officiels dont l'innocuité et la stabilité ne peuvent pas être garantis...

Dans le cas très précis de mon téléphone LG GW620, livré sous Android 1.5, LG France [17] a promis une mise à jour vers Android 1.6 pendant plus d'un an... avant de finalement se rétracter. Des firmwares Android 2.1 et 2.2 d'excellente facture – mais non officiels – ont donc été développés par des amateurs [18] ! Sous la pression de ses utilisateurs, LG a fini par sortir un firmware officiel... mais trop tard.

Fabricants de logiciels La concurrence fait rage dans le domaine des systèmes d'exploitation pour ordiphones.

Dès lors, les éditeurs de systèmes d'exploitation doivent séduire tous les autres acteurs, quitte à diminuer les contraintes de sécurité :

- Pour séduire les utilisateurs, il semble nécessaire de disposer du plus grand nombre d'applications dans sa MarketPlace.
  Bien qu'on puisse questionner cette logique (ne serait-il pas plus intéressant de disposer d'applications utiles, fiables et bien réalisées ?), la guerre des chiffres est néanmoins lancée.
- Pour séduire les développeurs d'applications, il est nécessaire de leur faciliter le travail au maximum.
  Le choix du langage Java et la mise à disposition d'un SDK gratuit pour Eclipse va dans ce sens. Mais Google va même plus loin en mettant à disposition un SDK

16. http://www.unrevoked.com/rootwiki/doku.php/public/unrevoked1_disclosure
17. http://twitter.com/lg_blog_france
18. http://openetna.com/openetna/

« visuel » pour les gens n'ayant aucune notion de développement [19]. On peut se poser des questions sur la sécurité (et la qualité en général) des applications qui en sortiront...
La signature d'applications fût également pendant longtemps une pierre d'achoppement. Lors du lancement du programme « Symbian Signed », on a pu lire que Symbian voulait « tuer » les développeurs indépendants. Heureusement, l'AppStore a depuis prouvé le contraire et Android n'a eu aucun mal à imposer la signature dès sa genèse.
– Séduire les fabricants de matériels est assez simple : il suffit de produire un système gratuit et d'en assurer une bonne partie du support. Ajoutez à cela une forte demande du marché, et tout est en place pour garantir le succès.

Circuits de distribution L'essentiel des ordiphones sont commercialisés par les opérateurs de téléphonie mobile (Google ayant d'ailleurs essuyé un échec commercial sur la vente « en direct » du Nexus One).

Comme la plupart des acteurs de la chaine, les opérateurs cherchent à maximiser leurs revenus... parfois au détriment de la sécurité :
– Les opérateurs personnalisent les systèmes d'exploitation ... en y ajoutant des bogues.
– Les opérateurs tentent de rendre payantes des fonctions natives... incitant de ce fait les utilisateurs au piratage.

Parmi les inventions géniales du marketing, on peut citer le bridage du *tethering* (utilisation du téléphone comme modem 3G), la géolocalisation payante (alors que le téléphone dispose d'un GPS intégré), l'impossibilité de regarder la télé ou d'écouter de la musique sur un lien WiFi (mais pas sur un lien 3G), etc.

Dès lors, les utilisateurs sont tentés de débrider leur téléphone (*jailbreaking*), d'installer des applications tierce partie douteuses, voire d'installer des systèmes non officiels... sans garantie d'innocuité.

Développeurs d'applications L'un des moyens les plus efficaces pour séduire les développements est... de leur promettre des revenus conséquents ! Pas facile dans un monde où la plupart des applications sont gratuites, ou dont le prix n'excède pas 1 .

Tout d'abord le processus d'achat d'applications est relativement simple grâce à Google Checkout, lorsque l'opérateur ne fournit pas lui-même le service de facturation au travers de sa MarketPlace. Pour les applications financées par la publicité, la réputation de Google dans le domaine n'est plus à faire (surtout depuis le rachat de la société AdMob, spécialisée dans le domaine de la publicité sur mobiles).

19. http://appinventor.googlelabs.com/about/

Contrairement à ce qu'on pourrait croire, les deux méthodes génèrent des revenus comparables, comme le révèle l'auteur d'une application inutile mais pourtant téléchargée presque 100 000 fois, à savoir Advanced Task Manager [20].

Les développeurs attirés par des revenus encore plus rapides pourront être tentés par le côté obscur de la téléphonie mobile : appels et SMS surtaxés, vol (et revente) du carnet d'adresses, collecte de l'historique du navigateur, etc.

Ce phénomène est loin d'être marginal, certaines applications étant clairement malveillantes – comme « SMS.AndroidOS.FakePlayer.a » ou les applications éditées par « 09Droid » qui se font passer pour des applications bancaires officielles – d'autres étant simplement à la limite du marketing agressif – comme celles de l'éditeur français Zeuzz.

Plusieurs études soulignent la dangerosité des MarketPlaces ; généralement financées par les éditeurs antivirus ces études ne sont toutefois pas d'une qualité scientifique suffisante pour être citées ici. Le risque semble néanmoins réel, puisque Google a retiré en urgence des dizaines d'applications de la MarketPlace dans le cadre de l'affaire « DroidDream » (nous y reviendrons).

4.2 Failles logicielles

État des lieux Compte-tenu de l'utilisation massive de code Open Source dans le système Android, les failles d'implémentation logicielle sont faciles à identifier : il suffit de suivre les alertes de sécurité des principaux projets ! Lorsqu'ils sont disponibles, ce qui n'est pas forcément le cas pour le noyau Linux...

Les vecteurs principaux (en termes de nombre de lignes de code et d'exposition) sont le système Linux et le navigateur Web (et particulièrement le moteur WebKit, commun avec Safari).

Il ne faut pas oublier non plus qu'Android intègre Flash Player... et vit avec les mêmes failles [21].

Failles navigateur Comme dans tout navigateur Web de complexité importante, les failles sont nombreuses. Ces failles n'en restent pas moins dangereuses, car elles ont été démontrées exploitables sur Android [22] [23].

L'une d'entre elles est même disponible dans Metasploit [24]. Il s'agit d'une faille relativement grave, qui permet de lire n'importe quel fichier du téléphone accessible

20. http://arronla.com/2010/08/android-revenue-advanced-task-manager/
21. http://www.theinquirer.net/inquirer/news/2034092/android-smartphones-attack-flash-player-hackers
22. http://www.exploit-db.com/exploits/15423/
23. http://www.exploit-db.com/exploits/16974/
24. http://blog.metasploit.com/2011/01/mobile-device-security-and-android-file.html

au processus navigateur - ce qui inclut l'intégralité de la carte SD, puisque celle-ci est formatée en FAT (donc sans aucun contrôle d'accès).

On notera dans cette attaque l'utilisation de l'URL `content://com.android.htmlfileprovider/<filename>`, qui permet de lire n'importe quel fichier du téléphone au travers d'un Content Provider [25] – une notion spécifique à Android qui promet d'être encore source de nombreuses failles, vu la surface d'attaque exposée.

Par ailleurs, outre les schémas traditionnels (`file://`, `http://`, etc.), Android supporte par défaut des schémas plus spécifiques et donc moins testés.

Parmi les schémas supportés, on peut citer `content://` (qui permet de communiquer avec un Content Provider), `market://` (qui permet d'interagir avec la MarketPlace, comme par exemple `market://details?id=<package>`) ou `android.resource://` (qui permet d'accéder à une ressource dans un paquet, comme par exemple `android.resource://<paquet>/<ressource>`).

Failles système Les failles système ne sont pas légion, mais ont toutes un impact critique sur la sécurité du système.

Parmi les failles les plus connues (car les plus exploitées « dans la nature »), on peut citer les failles publiées sur le site xSports [26], à savoir :

- `Exploid` : faille « udev » bien connue des administrateurs Linux, CVE-2009-1185.
- `Rage Against The Cage` : faille liée au mauvais comportement du processus `adb` lorsque `RLIMIT_NPROC` a été atteint [27] - l'appel à `setuid()` échoue mais le processus n'en tient pas compte et continue son exécution son l'identité `root`.
- Killing In The Name Of : faille permettant de modifier la valeur de la constante globale (normalement en lecture seule) `ro.secure`, en jouant sur une erreur d'implémentation de `/dev/ashmem`. Cette constante, définie à la compilation du système, indique au processus `adb` s'il doit s'exécuter sous l'identité `root` [28].

Les applications « légitimes » (comme VISIONary+ [29]) ou malveillantes (la plus célèbre ayant probablement été DroidDream [20]) ne se privent pas d'exploiter ces failles « dans la nature ».

Il faut toutefois noter qu'il existe une différence subtile entre obtenir un accès `root` et obtenir un accès `root` persistant au redémarrage du téléphone. En effet, bien qu'il soit virtuellement possible d'écrire dans le répertoire `/system` (au travers de la commande `adb remount` ou d'un `remount rw` à l'intérieur du terminal), un

25. http://developer.android.com/guide/topics/providers/content-providers.html
26. http://stealth.openwall.net/xSports/
27. http://dtors.org/2010/08/25/reversing-latest-exploid-release/
28. http://source.android.com/porting/build_system.html
29. http://android.modaco.com/content/software/320350/19-nov-r14-visionary-one-click-root/

certain nombre de téléphones modernes disposent d'une protection contre le *reflashage* intempestif de la partition système. Cette protection est souvent liée à l'utilisation d'un composant mémoire capable de vérifier une signature cryptographique (par exemple un composant eMMC [30]).

Dans le jargon, on dit que la possibilité de *reflasher* la partition système avec un firmware non signé nécessite un téléphone en mode S-OFF. Sur de nombreux téléphones de marque HTC [31], le mode courant (S-ON ou S-OFF) peut être obtenu en appuyant sur la touche *Volume Down* au démarrage du téléphone.

Failles logiques Il existe une catégorie de failles difficiles à détecter et pourtant dévastatrices : les failles logiques.

L'un des exemples les plus fameux est le déblocage du téléphone en utilisant son compte Google. Dans certains cas de corruption de données, ce processus peut échouer comme documenté officiellement par Google [32].

Dans ce cas, le téléphone doit normalement être réinitialisé en configuration d'usine (lorsque c'est possible – la procédure exacte dépend du constructeur). Heureusement, il s'avère que le mot de passe « null » est toujours accepté comme valide !

Un autre bogue célèbre affecte le Motorola Droid... et potentiellement d'autres. Lorsque le téléphone est verrouillé mais qu'un appel entrant arrive, il est possible d'appuyer sur le bouton *Back*, et ainsi d'arriver à l'écran d'accueil du téléphone [33].

Enfin une superbe faille logique a été découverte sur le site `https://market.android.com/`, qui permet d'installer à distance des applications sur un téléphone Android. Grâce à une faille de type XSS persistant sur ce site [34], il était possible de faire installer une application, puis de l'exécuter automatiquement, sur le téléphone de n'importe quel utilisateur. Un utilisateur pouvait être victime de cette attaque en cliquant sur un lien depuis son téléphone ou son PC (sous réserve d'être logué sous son compte Google).

La technique utilisée pour exécuter automatiquement l'application après son installation mérite d'être soulignée : elle consiste à déclarer dans le manifeste de l'application que celle-ci supporte un nouveau schéma d'URI (par exemple `trigger://`), puis à invoquer un lien `trigger://...` depuis la page Web d'origine.

Que peut-on faire ? Compte-tenu de la quantité de code C embarqué dans le système Android, il existe et il existera des bogues d'implémentation dans ce système,

30. `http://en.wikipedia.org/wiki/MultiMediaCard#eMMC`
31. Un site de référence pour les téléphones de marque HTC : `http://unrevoked.com/`.
32. `http://code.google.com/p/android/issues/detail?id=3006`
33. `http://techcrunch.com/2010/01/11/verizon-droid-security-bug/`
34. `http://jon.oberheide.org/blog/2011/03/07/how-i-almost-won-pwn2own-via-xss/`

conduisant pour certains à des failles de sécurité. WebKit et le système Linux sont des cibles de choix compte-tenu de la surface d'attaque exposée.

Toutefois, et contrairement à un système « classique », la mise à jour à grande échelle s'avère extrêmement problématique pour plusieurs raisons :

- Le facteur d'échelle (plusieurs millions d'unités en circulation rien qu'en France).
- La diversité et le haut degré de personnalisation des plateformes matérielles et logicielles.
- La propriété du terminal (lorsqu'il a été acheté par l'utilisateur).
- Les risques commerciaux en cas de blocage définitif du terminal (*brick*).

On peut raisonnablement considérer qu'en dehors du *geek* moyen, aucun utilisateur de smartphone sous Android n'a jamais mis à jour son système.

Les statistiques officielles de Google sur la répartition des ordiphones par version d'Android permettent de se faire une idée objective sur la vitesse de renouvellement du parc [35]. A l'heure où j'écris ces lignes, malgré la disponibilité du système Android 2.3, la majorité du parc se compose encore de systèmes Android 2.1 et 2.2 – sans compter sur les irréductibles 6% qui disposent d'une version antérieure. Je laisse aux statisticiens le soin de produire une étude plus détaillée, qui sera néanmoins biaisée par le fait que les données mondiales sont agrégées, tandis que les politiques de renouvellement des opérateurs sont très variables d'un pays à l'autre.

Dans ces conditions, il est loin le moment où un opérateur mobile fera du NAC/-NAP sur son réseau pour interdire la connexion de ordiphones qui ne sont pas à jour des correctifs.

Techniquement, le protocole FOTA (*Firmware Over-The-Air*) permet toutefois de mettre à jour à distance des terminaux. Ce protocole a déjà été utilisé à grande échelle lorsqu'une *backdoor* (exploitable à distance) a été découverte sur certains téléphones HTC de modèle Hero et Evo.

5 Audit d'applications Android

5.1 Auditer, pourquoi ?

L'audit sécurité d'applications tierces est une activité vieille comme le conseil en sécurité. Toutefois le circuit de distribution logicielle étant assez différent entre le monde du logiciel d'entreprise et les MarketPlaces pour mobiles, il est vrai que la question mérite d'être posée à nouveau.

A l'heure où j'écris ces lignes, Google n'effectue aucune vérification de sécurité sur les applications publiées dans l'Android Market. Cet état de fait pourrait changer,

35. http://developer.android.com/resources/dashboard/platform-versions.html

compte-tenu de la prolifération d'applications malveillantes, de plus en plus agressives et sophistiquées.

Parmi tous les risques liés aux terminaux mobiles (vol de l'équipement, intrusion via une faille du navigateur, etc.), on constate dans les faits que le risque principal aujourd'hui est la prolifération d'applications malveillantes. La « malveillance » est une notion assez floue, et certaines applications peuvent se situer dans une zone « grise » - comme l'application de voix sur IP Viber [36], qui duplique tous les contacts de l'utilisateur sur un serveur central faisant office d'annuaire téléphonique. Mais d'autres applications font l'unanimité contre elles, comme les applications utilisant des failles système pour élever leurs privilèges vers `root` et/ou envoyer des SMS surtaxés sans le consentement de l'utilisateur. Ces applications sont d'ailleurs « tuées » par Google une fois identifiées, car elles violent clairement les règles d'utilisation de l'Android Market.

Les circuits qui conduisent à l'apparition d'une application malveillante sont multiples :

- Malveillance délibérée de l'auteur qui veut gagner de l'argent rapidement. C'est le cas le plus simple. Un scénario courant consiste à « cloner » une application en vogue tout en y ajoutant du code malveillant.
- Compromission d'un éditeur d'applications à son insu. C'est la ligne de défense adoptée par plusieurs éditeurs d'applications après avoir été pointé du doigt. Cette défense est malheureusement crédible, car la plupart des développeurs sont des individus isolés ou des micro-entreprises attirées par les *success stories* « à la Angry Birds »... et bien souvent sans aucune sécurité dans les développements.
- Intégration de librairies tierce partie malveillantes dans un programme sain. L'auteur d'une application populaire (à savoir « Tank Hero ») a ainsi annoncé avoir été contacté par un fournisseur d'applications « probablement malveillantes », qui lui proposait un accord commercial [37].

Mais l'identification des applications malveillantes n'est pas la seule raison qui peut conduire à auditer une application Android. Parmi d'autres raisons « légitimes », on peut citer :

- Vérifier que l'application fait bien ce qu'elle prétend faire. Ce point est particulièrement critique pour les applications de sécurité, qui mettent en œuvre du chiffrement ou de la gestion de mots de passe par exemple. Dans le domaine, les surprises sont nombreuses, d'autant que les applications mobiles n'ont pas encore la maturité des applications « bureautiques », et que les contraintes

36. http://www.viber.com/
37. http://www.reddit.com/r/Android/comments/fm3cu/spyware_company_wants_us_to_embed_their_code_into/

du monde embarqué se font sentir (puissance de calcul, sources d'entropie, rémanence de l'allocateur mémoire Java, etc.).
– Recherche des failles de sécurité. Et oui, les applications Android sont aussi vulnérables aux *buffer overflows* lorsqu'elles utilisent du code natif (compilé avec le NDK), et même des injections SQL (même si SQLite n'implémente pas d'équivalent à `xp_cmdshell`). Il existe par ailleurs des attaques spécifiques au monde Android, comme les BroadcastReceiver qui ne vérifieraient pas la source des messages reçus.

Le problème principal avec l'audit d'applications Android n'est pas technique. Le problème c'est l'euphorie du marché : prolifération d'applications et mises à jour fréquentes, qui obligent à automatiser et industrialiser les audits [38].

5.2 Auditer, comment ?

L'audit sécurité est une tâche difficile à modéliser, car elle tire parti de la créativité de l'auditeur. On peut toutefois identifier les grandes lignes suivantes :

1. Récupérer l'application.
2. Décompresser les ressources.
3. Désassembler le bytecode.
4. Décompiler le bytecode (lorsque c'est possible).
5. Désassembler le code natif.
6. Déboguer l'application en cours d'exécution (sur un téléphone réel ou dans l'émulateur).

Cette méthodologie n'est pas exhaustive : elle ne donne pas accès au contenu téléchargé ultérieurement (par exemple au travers des publicités intégrées). Voyons les méthodes et les difficultés associées à chaque étape.

Récupérer l'application Cette étape n'est pas trop compliquée... pour qui est prêt à installer l'application cible sur son téléphone. Après installation, le fichier APK est conservé dans le répertoire `/data/app`. Selon les permissions appliquées au système de fichiers, il n'est pas nécessairement possible de lister le contenu de ce répertoire, mais les paquets restent lisibles par l'outil `adb`, et le nom de paquet peut être déterminé depuis la MarketPlace ou le menu « paramètres » du téléphone (le nom exact est `paquet-N.apk`, où N vaut généralement 1).

38. http://www.appanalysis.org/

Toute autre méthode est complexe... et probablement illégale. Parmi les méthodes possibles, on peut citer l'installation de l'application MarketPlace dans un émulateur, ou le *reverse engineering* de l'application MarketPlace à des fins de réimplémentation (cette dernière étant toutefois volumineuse, donc complexe à analyser).

Depuis la version 2.2 du système Android, et si l'application l'autorise, il est possible d'installer une application sur la carte SD. Mais l'application est alors stockée dans un fichier « `.asec` » chiffré. Ce fichier est inexploitable jusqu'à ce que le mécanisme et la clé de chiffrement soient connus.

On peut imaginer qu'une application extrêmement malveillante pourrait exploiter une faille dès l'installation pour élever ses privilèges vers `root`, et dissimuler ses traces (i.e. se rendre inaccessible de l'extérieur). Ce cas ne s'est pas encore produit « dans la nature » à l'heure où j'écris ces lignes, mais s'avère techniquement faisable.

Il faut noter que la MarketPlace de Google n'est pas la seule disponible, mais qu'il existe également des MarketPlaces d'opérateurs ou de tiers : ainsi Amazon commercialise désormais des applications Android [39]. Cette prolifération complexifie d'autant la tâche d'analyse des MarketPlaces. La méthode la plus « universelle » pour récupérer la source d'installation d'une application reste donc de l'installer sur un téléphone.

Un utilisateur peut installer des applications depuis des sources externes à la MarketPlace, si l'option « autoriser les sources inconnues » est cochée dans les paramètres de configuration. Il existe des applications légitimes accessibles en dehors des MarketPlaces, mais il existe également des bases d'applications « piratées » sur Internet, dont l'innocuité ne peut pas être garantie.

Notons que certaines applications, même parmi celles disponibles sur les MarketPlaces « officielles », exigent au préalable un accès `root` au téléphone !

Décompresser les ressources Les fichiers placés à l'intérieur d'un paquet APK sont optimisés d'une manière ou d'une autre avant d'être compressés. Ceci s'applique aussi bien au bytecode qu'aux ressources textuelles (comme les fichiers XML).

L'une des optimisations principales est la déduplication de données. Ainsi il n'existe qu'une seule instance de chaque chaine de caractères dans un fichier DEX. Différents encodages « optimisés » sont également utilisés, tel que LEB128 pour la représentation des entiers (*Little Endian Base 128*, aussi utilisé dans le format DWARF).

Toutes ces optimisations n'ont pas pour but la protection logicielle, et sont parfaitement réversibles. L'outil `aapt` fourni avec le SDK officiel permet ainsi d'explorer le contenu du manifeste d'un paquet APK. Les techniques d'optimisation ont été

39. http://www.amazon.com/appstore

analysées et documentées par des tiers, ce qui a permis la production d'outils de décompression comme `apktool` [40].

Désassembler le bytecode Avec le SDK officiel est fourni l'outil `dexdump`, qui permet de lister le bytecode contenu dans un fichier `CLASSES.DEX`.

Le format de fichier DEX a été lui aussi analysé et documenté [41], ce qui a permis la production de désassembleurs tiers comme `baksmali`.

Le code généré suit la syntaxe Jasmin [42], bien connue dans le monde Java « traditionnel ». Cette syntaxe peut sembler obscure de prime abord, mais elle est parfaitement régulière donc finalement très simple à appréhender. Prenons un exemple issu d'une application « réelle » : voici ci-dessous le code d'une méthode appelée `emptyResult`, telle que produit par `baksmali` :

```
    .method protected static emptyResult(Ljava/lang/String;)Z
        .locals 1
        .parameter "result"
        .prologue
        .line 306

        invoke-static {p0}, Lcom/utils/StringUtils;->isEmpty(Ljava/lang/String;)Z

        move-result v0
        return v0
    .end method
```

Toutes les directives commencent par un point. Leur nom est relativement explicite, `.locals 1` va par exemple indiquer la présence d'une seule variable locale. Le nom assigné aux *opcodes* de la machine virtuelle est également assez parlant.

`pN` représente l'argument numéro `N` passé à la méthode, `vN` représente le « registre » numéro `N` (Dalvik est une machine virtuelle à registres et non à pile).

Munis de ces informations, décortiquons la seule opération réellement complexe effectuée dans cet extrait de code.

40. http://code.google.com/p/android-apktool/
41. http://www.netmite.com/android/mydroid/dalvik/docs/dex-format.html
42. http://jasmin.sourceforge.net/

`invoke-static`	Invoque une méthode statique. Il existe également un *opcode* `invoke-virtual` pour les méthodes virtuelles, `invoke-super` pour la méthode de la superclasse, etc.
`{p0}`	La méthode appelée prend un seul argument. La valeur de cet argument provient du premier argument passé à la méthode `emptyResult`.
`L/com/utils/StringUtils;->isEmpty`	Invoque la méthode `isEmpty` de la classe `com.utils.StringUtils`. « `L` » est un préfixe accolé à tous les « littéraux ».
`(Ljava/lang/String;)`	Le premier (et seul) argument est de type `java.lang.String`.
`Z`	La méthode retourne un booléen.

A titre de référence, voici les différents types natifs disponibles dans Jasmin :

`Z`	Booléen
`B`	Octet
`C`	Caractère
`S`	Entier court (16 bits)
`I`	Entier (32 bits)
`J`	Entier long (64 bits)
`F`	Flottant (32 bits)
`D`	Double (64 bits)

L'avantage majeur de l'outil tiers `apktool` est la possibilité de modifier le bytecode ou les ressources d'une application, puis de la recompiler dans une version fonctionnelle. Ceci permet d'utiliser une technique d'analyse aussi ancienne que l'informatique, appelée « *printf debugging* ».

Dans le monde Android, l'utilisation de `System.out` doit être remplacée par `android.util.Log`. Cette classe définit plusieurs méthodes – respectivement `v`, `d`, `i`, `w`, `e` et `wtf` –, qui correspondent chacune à un niveau de « verbosité » – et dont les arguments sont enregistrés dans le journal « système » (sauf le niveau `debug`). Ce journal peut être consulté à l'aide de la commande `adb logcat`, apparait en temps

réel dans l'outil DDMS, et peut également être consulté par toute application qui dispose de la permission `READ_LOGS`.

Si l'on souhaite connaitre l'argument passé à la méthode `emptyResult` dans l'exemple précédent, il suffit alors d'ajouter un appel à n'importe quelle méthode de la classe `Log` de la manière suivante :

```
.method protected static emptyResult(Ljava/lang/String;)Z
    .locals 2
    .parameter "result"
    .prologue
    .line 306

    const-string v1, "PrintfDebuggingStyle"
    invoke-static {v1, p0}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/
        String;)I

    invoke-static {p0}, Lcom/utils/StringUtils;->isEmpty(Ljava/lang/String;)Z

    move-result v0
    return v0
.end method
```

Une variable locale `v1` a été ajoutée, contenant une chaine de caractères fixe. La méthode `d()` est ensuite invoquée. Cette modification est équivalente au code Java suivant :

```
Log.d("PrintfDebuggingStyle", <p0>);
```

On peut constater expérimentalement que de nombreuses applications de la MarketPlace font usage de l'API `Log`, ce qui permet de collecter des données de fonctionnement parfois très (trop) intéressantes.

Le journal « système » peut être collecté à distance par un développeur lors du *crash* de son application, si l'utilisateur l'autorise. Cette technique a été présentée par Renaud Lifchitz lors de la conférence 27c3 pour espionner les déplacements d'un utilisateur [43].

Décompiler le bytecode Comme avec tout bytecode de haut niveau (ex. Java, .NET), la sémantique de l'application est conservée.

Il est donc possible de décompiler du bytecode Dalvik et de reconstruire le code Java d'origine (sans les commentaires – seule information perdue à la compilation). Le premier outil public capable d'une telle opération fût `undx` [44] de Marc Schoenefeld. Cet outil est toutefois bogué et non maintenu par son auteur. Le meilleur outil disponible actuellement est l'outil `dex2jar` [45], qui reconstruit du bytecode Java à partir du

43. http://events.ccc.de/congress/2010/Fahrplan/events/4151.en.html
44. http://www.illegalaccess.org/undx.html
45. http://code.google.com/p/dex2jar/

bytecode Dalvik. Les outils « classiques » de décompilation Java [46] [47] s'appliquent alors.

Compte-tenu de la facilité déconcertante avec laquelle il est possible d'analyser les applications Android (et donc de les « *craquer* », entre autres), Google a mis à disposition des développeurs une bibliothèque très complète de gestion des licences en ligne : *Licensing Verification Library* (LVL) [48]. Toutes les versions de cette bibliothèque ont malheureusement été « *craquées* » elles aussi à l'heure où j'écris ces lignes...

Une autre contre-mesure mise en place par Google consiste à intégrer un obfuscateur de bytecode dans la chaine de compilation : ProGuard [49]. Cet outil relativement puissant est configurable par le développeur au travers du fichier `proguard.cfg` ; toutefois le fichier de configuration par défaut (reproduit ci-après) n'opère que le renommage des classes et méthodes invisibles à l'extérieur de l'application, ce qui est peu efficace face à un attaquant déterminé.

```
-optimizationpasses 5
-dontusemixedcaseclassnames
-dontskipnonpubliclibraryclasses
-dontpreverify
-verbose
-optimizations !code/simplification/arithmetic,!field/*,!class/merging/*
-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
-keep public class * extends android.content.BroadcastReceiver
-keep public class * extends android.content.ContentProvider
-keep public class * extends android.app.backup.BackupAgentHelper
-keep public class * extends android.preference.Preference
-keep public class com.android.vending.licensing.ILicensingService

-keepclasseswithmembernames class * {
    native <methods>;
}

-keepclasseswithmembernames class * {
    public <init>(android.content.Context, android.util.AttributeSet);
}

-keepclasseswithmembernames class * {
    public <init>(android.content.Context, android.util.AttributeSet, int);
}

-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}
```

46. http://members.fortunecity.com/neshkov/dj.html
47. http://java.decompiler.free.fr/
48. http://developer.android.com/guide/publishing/licensing.html
49. http://developer.android.com/guide/developing/tools/proguard.html

```
-keep class * implements android.os.Parcelable {
  public static final android.os.Parcelable$Creator *;
}
```

Au vu des résultats obtenus, on peut considérer que l'obfuscation de bytecode Java dans le monde Android est en retard sur l'obfuscation de code x86 dans le monde PC. Les techniques de complexification du flot de contrôle ou de construction dynamique des constantes ne sont pas encore répandues dans les applications commerciales pour Android. Afin de combler ce manque, des projets parallèles (comme AndroGuard [50]) fleurissent - mais on peut supposer que la solution intégrée au SDK fourni par Google restera la solution majoritairement utilisée par les applications (pour des raisons de simplicité évidentes).

Désassembler du code natif Lors de l'audit d'une application Android, il est possible de rencontrer du code natif dans deux cas :

1. L'application contient des bibliothèques natives, compilées avec le NDK.
2. L'application contient des applications natives, ou s'avère être elle-même une application native. Ce cas est courant avec les applications malveillantes, qui embarquent du code natif pour exploiter des failles noyau [51].

Dans les deux cas, Android ne présente aucune spécificité par rapport à un Linux/ARM classique. Les fichiers natifs sont au format ELF. La chaine de compilation native est construite autour du compilateur GCC. Tous les outils sont disponibles depuis longtemps pour analyser des tels binaires (ex. IDA Pro, Hex-Rays Decompiler ARM, etc.).

Déboguer l'application Une application Java compilée en mode « *debug* » peut être déboguée à distance au travers du protocole standard *Java Debug Wire Protocol* (JDWP), défini dans le cadre de *Java Platform Debugger Architecture* (JPDA).

Les environnements de développement Java « classiques » (comme Eclipse) permettent ainsi de déboguer une application à distance une fois celle-ci installée sur le téléphone.

Il est également possible d'utiliser le vénérable JDB (fourni avec le SDK Java [52]), bien que cet outil n'ait jamais dépassé le statut de « preuve de concept » à mon avis.

L'outil DDMS, fourni avec le SDK Android, repose également sur JPDA. Cet outil ne permet toutefois pas le débogage d'applications, mais seulement l'analyse de performance.

50. http://code.google.com/p/androguard/
51. http://stealth.openwall.net/xSports/
52. http://download.oracle.com/javase/1.4.2/docs/tooldocs/solaris/jdb.html

La liste des applications pouvant être déboguées sur un système Android s'obtient avec la commande `adb jdwp`.

Le vrai problème se pose pour les applications qui n'ont pas été compilées en mode « *debug* ». Ceci implique que le manifeste de l'application ne contient pas d'entrée `<application android:debuggable="true">`. De plus la plupart des outils de débogage Java nécessitent que le bytecode contienne des indications sur les numéros de lignes dans le code source d'origine (il s'agit d'une limitation arbitraire qui ne s'applique pas à l'outil JavaSnoop [53] par exemple).

Fort heureusement, il est possible de désassembler ces applications, puis de les réassembler en mode « *debug* » avec l'outil `apktool` [54]. Le processus est complexe et le résultat peu convivial, mais il a le mérite d'exister.

Pour réinstaller l'application modifiée sur un équipement « réel », il est nécessaire de la signer à nouveau. La procédure est simple, et parfaitement documentée par Google.

Il faut d'abord générer une clé personnelle autosignée, à durée de vie extrêmement longue :

```
$ keytool -genkey -v -keystore mykey.keystore -alias nom_alias -keyalg RSA -
    keysize 2048 -validity 10000
```

Il faut ensuite signer le nouvel APK avec les outils du SDK Java :

```
$ jarsigner -verbose -keystore mykey.keystore nom_application.apk nom_alias
```

Enfin il est recommandé d'aligner le code sur un multiple de 4 octets pour des raisons de performance :

```
$ zipalign -v 4 nom_application.apk nom_application_final.apk
```

Instrumentation Il est possible d'instrumenter une application Java à l'aide de classes spécifiques à Android (`android.app.Instrumentation.*` [55]). Cette méthode fournit des résultats détaillés et personnalisables sur le fonctionnement interne d'une application, elle nécessite toutefois une décompilation et une modification préalable de l'application à auditer.

A l'heure où j'écris ces lignes, il n'existe pas (à ma connaissance) d'outil permettant de généraliser et d'automatiser le processus à tout fichier APK. Cette voie de recherche pourrait toutefois donner des résultats intéressants.

53. http://www.aspectsecurity.com/tools/javasnoop/
54. http://code.google.com/p/android-apktool/wiki/SmaliDebugging
55. http://developer.android.com/reference/android/app/Instrumentation.html

Outils spécifiques Les outils disponibles actuellement dans le monde Android sont rudimentaires, quoi que suffisants.

Toutefois on peut raisonnablement estimer que la quantité et la qualité des outils disponibles va croitre de concert avec la popularité du système Android.

A l'heure où j'écris ces lignes, la version 6.1 du logiciel IDA Pro [56] vient de sortir. Cette version ajoute le support du format DEX et du bytecode Dalvik, ainsi que le débogage à distance des applications Android natives.

On peut supposer que d'autres outils couramment utilisés par les auditeurs en sécurité vont être mis à jour pour s'adapter au monde Android.

6 Conclusion

Le système Android n'a malheureusement pas profité de l'opportunité unique qui lui était donnée : repartir de la feuille blanche et créer un système d'exploitation « sûr par conception ».

A cause d'un héritage logiciel peu fiable, ainsi que des contraintes matérielles et commerciales issues du monde de la téléphonie mobile, le système résultant est un compromis qui laisse une large place à la créativité des attaquants.

Celle-ci est d'autant plus exacerbée que la téléphonie mobile est un phénomène de masse (il y a plus de téléphones mobiles que d'ordinateurs dans le monde) et que le téléphone mobile est par conception un système de paiement (ce que n'est pas un PC traditionnel).

A ce tableau vient s'ajouter 30 ans d'expérience dans le domaine des attaques logicielles, appliquées à un écosystème applicatif qui n'existait pas il y a 5 ans.

Toutes les conditions sont donc réunies pour que des attaques de grande ampleur, motivées par l'argent, se produisent contre les matériels grand public équipés du système Android (téléphones, tablettes, télévisions, ...).

La bonne nouvelle pour l'expert en sécurité, c'est qu'il est encore temps de s'y mettre ! Android est suffisamment jeune et accessible aux nouveaux entrants, ce qui n'est plus le cas de la plateforme PC, où les techniques d'attaque (ex. exploitation d'un *heap overflow*) et de défense (ex. obfuscation logicielle) ont atteint des sommets de raffinement désormais accessibles aux seuls experts les plus pointus du domaine.

7 Sites et outils

7.1 Sites officiels

– Android, portail officiel : http://www.android.com/

56. http://www.hex-rays.com/idapro/

- Android, MarketPlace officielle : https://market.android.com/
- Android, référence des développeurs : http://developer.android.com/

7.2 Sites non officiels

- Référence de la machine virtuelle Dalvik : http://dalvikvm.com/

7.3 Outils tiers

- Google Android Debugging Utilities : http://sites.google.com/site/ortegaalfredo/android
- Undx : http://www.illegalaccess.org/undx.html
- Smali/Baksmali : http://code.google.com/p/smali/
- Dedexer : http://dedexer.sourceforge.net/
- Dex2Jar : http://code.google.com/p/dex2jar/
- TaintDroid http://www.appanalysis.org/
- AndroGuard : http://androguard.blogspot.com/

Références

1. Benjamin Morin. Modèle de sécurité d'Android. MISC 51, 2010.
2. K. Chen. Reversing and exploiting an apple firmware update. http://www.blackhat.com/presentations/bh-usa-09/CHEN/BHUSA09-Chen-RevAppleFirm-PAPER.pdf BlackHat, 2009.
3. Alexandre Gazet. Sticky fingers & KBC Custom Shop http://www.sstic.org/2011/presentation/sticky_fingers_and_kbc_custom_shop/ SSTIC, 2011.
4. Loïc Duflot and Yves-Alexis Perez. Quelques éléments en matière de sécurité des cartes réseau http://www.sstic.org/2010/presentation/Peut_on_faire_confiance_aux_cartes_reseau/ SSTIC, 2010.
5. Alexander Tereshkin and Rafal Wojtczuk Introducing Ring -3 Rootkits http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf BlackHat USA, 2009.
6. Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. Run-time firmware integrity verification : what if you can't trust your network card ? CanSecWest, 2011.
7. Ralf-Philipp Weinmann. All Your Baseband Are Belong To Us. http://events.ccc.de/congress/2010/Fahrplan/events/4090.en.html CCC, 2010.
8. Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima et Toshiaki Tanaka. A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework SecureCom'10
9. Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima et Toshiaki Tanaka. A Small but Non-negligible Flaw in the Android Permission Scheme Wook Shin, Sanghoon Kwak, Shinsaku Kiyomoto, Kazuhide Fukushima et Toshiaki Tanaka IEEE POLICY 2010
10. Marc Schönefled. Reconstructing Dalvik Applications http://cansecwest.com/csw09/csw09-schoenefeld.pdf CanSecWest, 2009

11. Sergio Alvarez. The Smart-Phones Nightmare (iPhone-oriented, parts about Android) http://cansecwest.com/csw09/csw09-alvarez.pdf CanSecWest 2009
12. Nicolas Economou, Alfredo Ortega. Smartphone (in)security http://cansecwest.com/csw09/csw09-ortega-economou.pdf CanSecWest 2009
13. Jesse Burns. Exploratory Android Surgery http://www.blackhat.com/presentations/bh-usa-09/BURNS/BHUSA09-Burns-AndroidSurgery-SLIDES.pdf BlackHat USA 2009
14. Kevin Mahaffey, John Hering. App Attack https://media.blackhat.com/bh-us-10/presentations/Mahaffey_Hering/Blackhat-USA-2010-Mahaffey-Hering-Lookout-App-Genome-slides.pdf BlackHat USA 2010
15. Nils. The Risk you carry in your Pocket https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-slides.pdf BlackHat Abu Dhabi 2010
16. Raphaël Rigo. Android : Reverse Engineering and Forensics https://deepsec.net/docs/speaker.html#PSLOT45 DeepSec 2010
17. Itzhak Avraham. Popping Shell on A(ndroid)RM Devices https://media.blackhat.com/bh-dc-11/Avraham/BlackHat_DC_2011_Avraham-Popping_Android_Devices-Slides.pdf BlackHat DC 2011
18. Jon Oberheide http://jon.oberheide.org/blog/2010/06/25/remote-kill-and-install-on-google-android/
19. Jon Oberheide http://jon.oberheide.org/blog/2010/06/28/a-peek-inside-the-gtalkservice-connection/
20. http://www.androidpolice.com/2011/03/01/the-mother-of-all-android-malware-has-arrived-stolen-apps-released-to-the-market-that-root-your-phone-steal-your-data-and-open-backdoor/