Application de l'analyse de la mémoire vive

Ce chapitre propose une technique d'analyse de la mémoire vive visant un équilibre entre une approche suffisamment spécifique pour capturer des comportements malicieux particuliers à la plateforme Android, mais assez générique pour demeurer valide à travers les versions du SE. La méthode proposée utilise les greffons de *Volatility* ciblant l'extraction de structures internes du noyau Linux depuis une capture de la mémoire vive et les applique dans un contexte d'analyse de maliciels sur Android. Il est également proposé d'utiliser une technique proposée dans ce mémoire, l'analyse différentielle de la mémoire vive (ADMV), afin de concentrer l'effort d'analyse sur les structures affectées par l'exécution de l'application. Ce chapitre présente en premier lieu le rationnel et le fonctionnement de l'ADMV. En second lieu, il présente une étude dans laquelle l'approche est appliquée afin de capturer et d'extraire des artefacts de l'exécution d'un maliciel expérimental conçu à cette fin et vise à démontrer la pertinence de l'ADMV sur une plateforme Android utilisant ART.

4.1 Analyse différentielle de la mémoire vive

L'analyse différentielle peut être définie comme l'utilisation de la différence entre les données recueillies à deux temps de mesure t espacés par Δt. La résultante permet d'isoler uniquement les éléments ayant été créés, supprimés ou modifiés entre ces temps de mesure. Ce substrat réduit la quantité de données à analyser et vise à réduire le temps et la complexité de d'analyse. Dans le cas de l'analyse de maliciels, il est possible d'obtenir un état initial qui se définit par l'état de la plateforme au moment qui précède le déploiement de l'échantillon à analyser. Pour l'ADMV, cela signifie qu'il est possible d'obtenir une acquisition à un temps t_{pre} où l'application à analyser n'a pas encore été déployée et que le système fonctionne normalement. Après l'acquisition de cette capture initiale, l'application est déployée. Afin d'augmenter les chances qu'elle se révèle malicieuse si tel est le cas, elle peut être stimulée manuellement par

les actions d'un analyste, automatiquement par des outils simulant un utilisateur ou par des évènements divers survenant sur la plateforme (p. ex. réception de message texte, de courriel, délais, etc.). Une seconde acquisition est alors effectuée à un temps post-déploiement t_{post} où les artefacts de l'exécution de l'application sont possiblement encore présents dans la mémoire vive.

Afin d'effectuer la comparaison entre les deux temps de mesure, il est proposé de comparer les informations extraites à l'aide des modules visant Linux de *Volatility*. Tel qu'il est décrit précédemment, *Volatility* permet d'extraire des informations sur les structures internes du SE à partir de la mémoire vive. La comparaison des résultats obtenus par l'outil permet d'établir clairement quelles composantes du SE sont d'intérêt pour l'analyse. Il est postulé que les nouvelles entrées sont des éléments ayant été créés pendant l'exécution de l'application. Les éléments présents avant et après le déploiement de cette dernière sont considérés pour l'analyse s'ils ont été modifiés dans l'intervalle d'acquisition. Finalement, les éléments présents au temps t_{pre} et absent au temps t_{post} sont classés comme ayant été supprimés. Cette approche ne permet pas de capturer les artefacts ajoutés à la mémoire vive après le temps t_{pre} et retirés avant t_{post}.

L'ADMV a également été explorée par 504ensics Labs [2] en visant principalement les greffons Windows de *Volatility*. Leur approche est démontrée par l'outil Differential Analysis of Malware in Memory (DAMM) qui vise à automatiser la différenciation entre les captures de mémoire vive. Ce programme a été étudié dans les présents travaux et des efforts ont été menés afin de l'adapter aux greffons pour Linux de *Volatility*. Toutefois, le développement de l'outil a été abandonné par son créateur et les efforts pour le maintenir ont été interrompus. Conséquemment, DAMM n'a pas été retenu dans les présents travaux bien qu'il en démontre l'intérêt de l'approche.

Dans l'état actuel de l'approche proposée, un analyste est requis afin de filtrer les résultats de l'ADMV. En effet, le différentiel obtenu comprend tous changements apportés par l'application analysée, autant ceux apportés par la cible de l'analyse que ceux dus au fonctionnement normal du SE se produisant pendant la durée de vie de l'analyse. L'étude décrite à la section suivante a été réalisée afin d'évaluer si l'ADMV permet de cerner correctement les comportements malicieux d'une application en suivant les principes énoncés.

4.2 Étude de cas

L'approche de l'ADMV a reçu peu d'attention dans la littérature. Sur Android, il n'a pas été possible d'identifier des travaux s'étant penchés sur sa pertinence et son utilisation en tant qu'outil d'analyse de maliciels. C'est pour cette raison que l'étude suivante s'est intéressée à démontrer l'applicabilité de l'ADMV dans l'identification de comportements malicieux de même que sa résilience aux tactiques d'évasion de la détection comme l'obfuscation, le

chargement dynamique de code malicieux et l'élévation de privilège sur une version d'Android utilisant ART. La version 5.1 a été sélectionnée à cet effet.

Plus spécifiquement, l'expérimentation décrite vise à répondre aux questions suivantes : est-ce possible d'utiliser l'ADMV pour identifier, récupérer et analyser le contenu d'un fichier de code malicieux déchiffré, chargé dynamiquement et supprimé dès la fin de son chargement afin d'empêcher sa récupération sur le stockage physique de l'appareil ? Également, est-il possible par cette même technique d'identifier une élévation de privilèges et un cheval de Troie (ou porte dérobée) ?

Dans un premier temps, les aspects méthodologiques de l'étude sont présentés, suivis, dans un deuxième temps, des résultats obtenus. Dans un troisième temps, une discussion présente l'intérêt de ces résultats. Finalement, les limitations de l'étude ainsi que les pistes de recherche futures sont présentées.

4.3 Méthodologie

Un maliciel a été spécifiquement conçu pour manifester les comportements suivants visés par l'étude soit :

- l'obfuscation de ressources (code) malicieuses (du code malicieux dans le cas présent) ;
- le déchiffrement et chargement dynamique de code externe ;
- la suppression des fichiers intermédiaires ;
- l'exécution de ressources binaires externes ;
- l'élévation de privilèges et
- la connexion à distance à un serveur de commande et contrôle.

Ce maliciel expérimental couvre plusieurs problématiques de l'analyse de maliciels qui peuvent représenter un défi pour les méthodes d'analyse statique et d'analyse dynamique décrites au chapitre 2. Le maliciel en apparence bénin est déployé sur un téléphone intelligent Android muni d'une capacité d'acquisition de la mémoire vive afin de capturer l'activité suspecte.

L'appareil utilisé pour le déploiement de l'application est un téléphone Nexus 4 de LG avec la version 5.1 du système d'exploitation. Ce dernier a été recompilé pour supporter le chargement du module LiME selon la méthode présentée à la section 3.1.1. Le module est préchargé sur l'appareil avant le scénario. Également, pour le bien de la démonstration, le noyau a été modifié afin d'être vulnérable à l'élévation de privilèges utiliser dans l'exploit *towelroot* introduit par Hotz [63]. La révision du noyau utilisée correspond à une variante du noyau Linux, appelée Mako, et adaptée au Nexus 4. La version du code utilisée se situe sur la branche Git appelée *origin/android-msm-mako-3.4-lollipop-mr1*. La seule exception est le fichier *kernel/futex.c* qui est à une version antérieure, soit à la révision identifiée par l'identifiant unique Git *7e9543a0ebde3c1df0108523a3e9e152e254f962*. Cette dernière précède le correctif appliqué pour

protéger des exploits utilisant la même technique que *towelroot*. Cette modification a pour effet de retourner le noyau Linux à une version vulnérable tout en restant compatible avec la version 5.1 d'Android. Le fonctionnement de cet exploit est détaillé par Zatuchna sur *Nativeflow* [134–136]. La version utilisée dans le cadre de cette étude est celle de Timwr [120]. Finalement, l'utilisateur *root* est activé à l'aide du correctif SuperSU conçu par Chainfire [22] afin de permettre l'acquisition par LiME. Toutefois, l'utilisation de ces privilèges n'est limitée qu'à l'invite de commande accessible par l'interface de déverminage ADB et utilisée pour lancer l'acquisition.

L'application Android utilisée est une adaptation de celle proposée par Timrae [119]. Dans sa version d'origine, elle démontre le chargement dynamique d'un fichier de type Java Archive (JAR), une archive Java contenant un fichier Android « classes.dex » renfermant du code pouvant être chargé par une application. À l'origine, la version de Timrae n'est pas malicieuse. Pour l'étude, elle a été adaptée afin que le contenu du JAR soit chiffré et puisse être déchiffré à la volée avant d'être chargé dynamiquement. La figure 4.1 présente l'interface de l'application malicieuse. La figure 4.2 illustre la notification produit par le chargement dynamique.

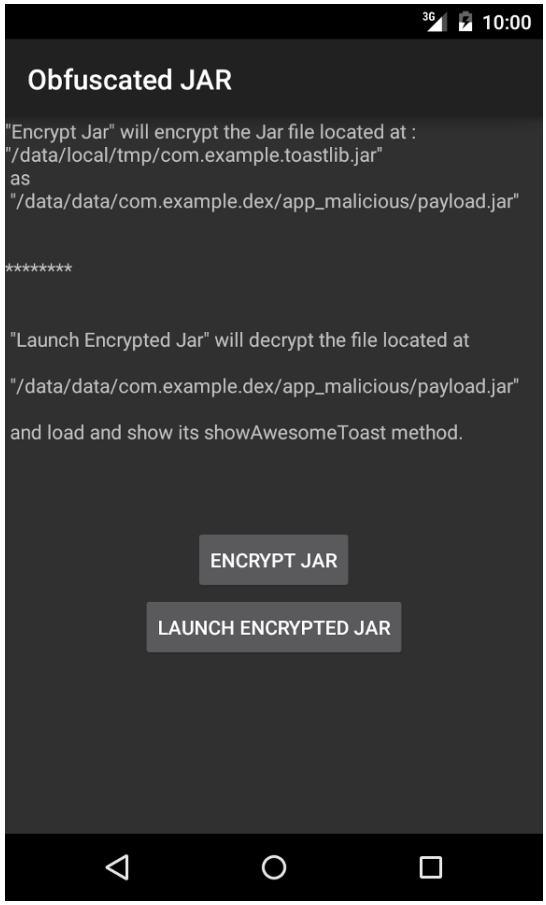

FIGURE 4.1 – Écran d'accueil de l'application.

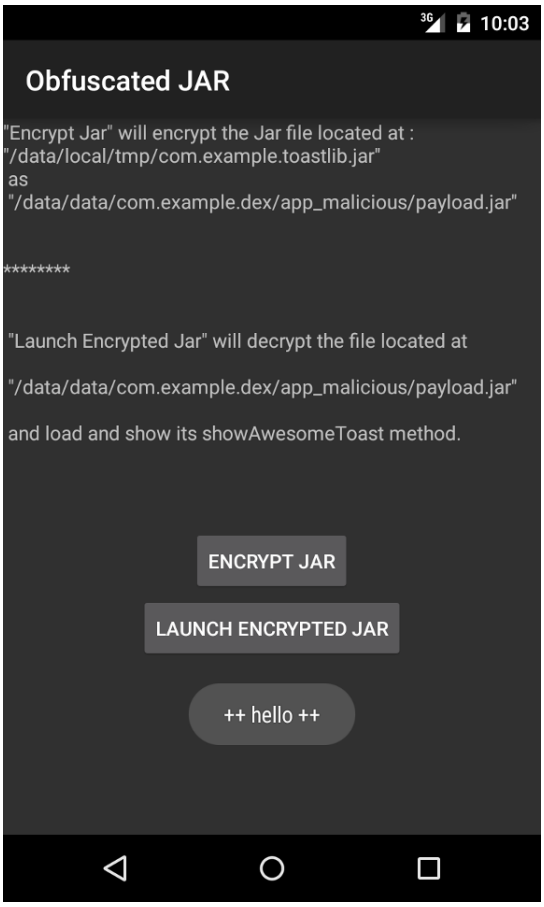

FIGURE 4.2 – Notification résultant du chargement dynamique de la ressource chiffrée.

D'autres modifications ont été apportées afin de représenter des comportements malicieux et d'évasion de la détection. Notamment, le programme récupère 3 fichiers depuis une source externe. Ces 3 fichiers sont le JAR contenant le code malicieux à exécuter, l'exploit d'élévation de privilège sous la forme de l'exécutable *towelroot* et l'utilitaire Linux *busybox*, comportant plusieurs programmes élémentaires n'ayant pas été inclus dans la version du noyau Linux d'Android. Pour simplifier le déploiement, l'emplacement d'où ces fichiers sont récupérés est, dans le cadre de cette étude, le dossier présent sur les systèmes Android à l'emplacement « */data/local/tmp* ». Ils sont chargés manuellement par l'expérimentateur avant l'exécution du scénario. Dans un contexte réel de maliciel, ces fichiers seraient probablement récupérés sur un serveur distant pour réduire le risque de détection au moment de l'installation. Ce choix méthodologique n'affecte pas la pertinence de l'étude puisque le chargement de ces ressources depuis un emplacement local correspondrait à l'étape suivant le téléchargement depuis un serveur distant.

Le flot d'exécution suit le modèle ci-dessous :

1. Le bouton « Encrypt JAR » permet à l'application de faire une copie chiffrée des 3

fichiers dans un espace de stockage réservé pour ses besoins uniquement, soit un dossier local situé à l'emplacement « */data/data/com.example.dex/* ». Le contenu est placé dans le sous-dossier « *app_malicious* ». Cette opération simule le téléchargement des fichiers chiffrés sur l'appareil et n'est effectuée que pour la configuration après l'installation du maliciel. Son principal but est de simplifier la manipulation par l'expérimentateur. Les fichiers sont chiffrés par le passage d'une opération binaire de type ou-exclusif (XOR) avec une clef correspondant à $0xAA$ en base hexadécimale. Cette opération n'est pas adaptée pour protéger le contenu des données, mais elle sert à illustrer avec simplicité qu'il est possible de chiffrer et déchiffrer du code à la volée.

2. Le bouton « Launch Encrypted JAR » est utilisé pour lancer le comportement malicieux. Lorsqu'il est appuyé, une méthode charge les fichiers chiffrés acheminés dans le dossier local à l'application, les déchiffre par le passage d'un XOR avec la même clef hexadécimale $0xAA$ et charge dynamiquement le contenu du JAR. Ce dernier effectue le comportement suivant.
   a) Il affiche la notification avec le message « ++ hello ++ » tel qu'illustré à la figure 4.2 .
   b) Cette étape correspond au moment où le comportement est jugé malicieux. Le maliciel lance l'exploit *towelroot* qui est utilisé pour obtenir les accès *root*. Avec ces privilèges, *busybox* est utilisé pour établir un serveur *netcat* à l'écoute sur le port 5550. Ce dernier relaye toutes les commandes reçues à une invite de commande *sh* lancée sur l'appareil Android avec les permissions *root* et retransmet au client distant tous les messages générés par les commandes envoyées. D'un point de vue pratique, le comportement décrit s'apparente à celui d'un cheval de Troie ayant les privilèges de l'utilisateur *root* sur l'appareil.
   c) Une fois le cheval de Troie déployé, le code chargé dynamiquement procède à la suppression de tous les fichiers déchiffrés ainsi que les fichiers intermédiaires engendrés par le chargement dynamique du JAR. Cette étape a pour but d'en empêcher la récupération par un analyste depuis le stockage de l'appareil.
   d) En arrière-plan, le cheval de Troie est libre de poursuivre son exécution puisque les fichiers nécessaires à son fonctionnement demeurent chargés en mémoire tant que l'application s'exécute.

L'acquisition de la mémoire vive est effectuée avec LiME. L'exécution d'une analyse débute après avoir chiffré les ressources à l'aide du bouton « Encrypt JAR » et se déroule de la façon suivante :

1. L'appareil est redémarré afin de vider le contenu de la mémoire vive et d'éviter le bruit dans l'acquisition. Rien d'autre n'est effectué avant que la plateforme et tous ses services ne soient démarrés.

2. L'application *Obfuscated JAR* est lancée.
3. La première acquisition de la mémoire vive est réalisée avec LiME.
4. Lorsque l'acquisition est complétée, l'utilisateur appuie sur « Launch Encrypted JAR ».
5. À l'aide d'un client *netcat*, une connexion est établie entre l'appareil et l'attaquant.
6. Une seconde acquisition de la mémoire vive est réalisée avec LiME.
7. À la fin de l'acquisition, l'expérimentation prend fin.

Volatility est utilisé pour analyser la capture. Il vise à récupérer les fichiers intermédiaires non chiffrés produits par le chargement dynamique et ayant été supprimé du stockage physique pendant l'analyse. Il est utilisé dans le but de récupérer toutes traces de l'exécution du cheval de Troie. Pour ce faire, une sélection de greffons *Volatility* visant le noyau Linux est retenue pour leur capacité à cibler les comportements malicieux en cause dans l'étude. Ces modules sont listés ci-dessous avec une justification de leur utilisation.

LINUX_PSTREE liste les processus en exécution au moment de l'acquisition ainsi que les relations parent-enfant entre ceux-ci. Une contribution au module proposé par Lebel et Ouellet [75] permet de récupérer également l'ensemble des informations contenues dans la structure *thread_info* et permet de lire les champs de l'identifiant du processus parent (Parent Process Id, PPID), l'identifiant d'utilisateur (User Id, UID), l'identifiant de groupe (Group Id, GID) et l'identifiant d'utilisateur effectif (Effective User Id, EUID). Ces informations permettent de savoir si de nouveaux processus sont créés pendant le temps d'analyse. Elles permettent également de valider si certains processus, nouveaux ou déjà présents, ont obtenu des privilèges *root*. Pour ce faire, les valeurs des champs UID, GID et EUID sont observées. Si la valeur de ceux-ci est 0, cela signifie que le processus s'exécute avec ces privilèges et peut constituer un marqueur de compromission.

LINUX_THREADS permet de répertorier l'ensemble des fils d'exécution présents au moment de la capture, leur processus d'appartenance, ainsi que les valeurs de UID, GID et EUID rattachées. Il est possible de connaître les fils d'exécution créés par une application et de voir toutes incongruités dans les privilèges de ceux-ci.

LINUX_PSAUX liste également les processus en exécution à l'instar de LINUX_PSTREE. Il fournit également l'ensemble des arguments utilisés pour lancer le processus. L'utilité de ce module est donc de pouvoir récupérer les commandes suspectes utilisées dans le lancement d'un processus. Cela permet de capturer les comportements de maliciels, comme celui utilisé dans l'étude, qui ont recours à l'invite de commande pour exécuter certaines actions comme démarrer un serveur pour recevoir les commandes distantes.

LINUX_ELFS liste l'ensemble des fichiers Linux exécutables (Executable Linux File, ELF) chargés en mémoire vive au moment de la capture. Ces fichiers contiennent des instructions binaires pouvant être exécutées directement par le SE ou utilisées à titre de ressources partagées par d'autres applications. Si des fichiers de type ELF sont chargés pendant l'analyse, il s'agit d'un indicateur d'utilisation de code natif dans le contexte de l'analyse. Plus précisément, cela peut indiquer que l'application utilise des bibliothèques logicielles natives du SE ou encore celles d'autres tiers. Ce type de comportement est particulièrement d'intérêt puisqu'il constitue un angle mort de plusieurs analyses recensées au chapitre 2.

LINUX_ARP extrait l'état de la table de routage au moment de la capture. Il est utilisé afin d'observer si les entrées ont changé dans cette table et inférer la présence de nouvelles connexions pendant l'exécution. De cette façon, il est possible d'établir si des connexions ont été effectuées pendant la durée de vie de l'analyse et de valider si celles-ci mènent à une source malicieuse. Ce module est utilisé en remplacement de LINUX_NETSTAT qui vise spécifiquement à récupérer les connexions réseau existantes dans le SE au moment de la capture. En effet, lors de tests préliminaires, il s'est avéré que ce dernier ne fonctionnait pas correctement sur Android et ne retournait aucune information lors de son exécution.

LINUX_FIND_FILE liste l'ensemble des fichiers ouverts par le système d'exploitation (SE) au moment de la capture. Cela permet de vérifier quels sont les fichiers utilisés par les processus, de valider si certains fichiers d'intérêt (p. ex. fichier ELF ou DEX) sont présents en mémoire vive et, le cas échéant, de les extraire dans leur intégralité pour analyse approfondie.

Il est postulé que l'ensemble de ces greffons permettent de reconstituer l'action malicieuse de l'application et de récupérer des artefacts de son exécution, même si ceux-ci sont supprimés de l'espace de stockage afin d'éviter la détection. Les résultats sont présentés à la section suivante.

4.4 Résultats

Deux fichiers contenant l'entièreté de la mémoire vive ont été recueillis, l'un avant l'opération de déchiffrement et du chargement dynamique et l'autre après. Les greffons *Volatility* sélectionnés ont été exécutés afin d'extraire les traces de l'exécution du maliciel. Les résultats obtenus pour l'analyse sur les captures au temps t_{pre} et t_{post} sont comparés afin d'extraire les éléments qui diffèrent. Les résultats de ces comparaisons sont présentés par modules exécutés ci-après.

LINUX_PSTREE : Les éléments d'intérêt résultant du différentiel entre les deux moments d'acquisition sont présentés au listage 4.1.

```
1 Offset          Name              PID   PPID  UID   GID   EUID
2 0xe5306000      init              1     0     0     0     0
3 0xec3ee800      com.example.dex   2849  203   10087 10087 10087
4 0xec8fdc00      towelroot-dec     3451  2849  0     0     0
5 0xec917c00      towelroot-dec     3465  0     0     0     0
```

Listing 4.1 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_pstree*.

Sur la plateforme Linux, un PPID de valeur 0 est attribué au premier processus (*init*) et tous les autres sont des enfants directs (relation parent-enfant) ou indirects (plusieurs relations parent-enfant imbriquées) de celui-ci. Or, il apparaît que le processus *towelroot-dec* avec le PID 3465 ne respecte pas cette règle. Également, il s'exécute sous l'UID, le GID et l'EUID 0, correspondant à l'utilisateur *root* et les privilèges qui s'y rattachent. Le processus 3451 s'exécute lui aussi sous cet utilisateur. Ces éléments soulignent le caractère anormal de ce processus puisque sur un SE d'Android non modifié, l'utilisateur *root* n'est utilisé que pour le processus *init* ($PID == 1$), *kthreadd*($PID == 2$) et les enfants directs de ceux-ci ($PPID == [1,2]$). À noter qu'il se peut que des modifications du SE par l'utilisateur ou le fabricant créent des exceptions à cette règle. Dans ces cas, elle peut être adaptée à la plateforme en question pour y inclure ces exceptions légitimes. La sortie brute en annexe F de la capture *a priori* rapporte l'ensemble des processus s'exécutant sur la plateforme où il est possible d'observer cette relation. Dans le cas de la présente étude, comme il est expliqué par Zatuchna [134, 135, 136], l'exploit *towelroot* utilise une chaîne de vulnérabilités afin d'arriver à altérer les propriétés PID, PPID, UID, GID et EUID de son propre processus directement dans la mémoire vive lui permettant de s'élever en tant qu'utilisateur *root*. Cette modification viole la relation décrite et est vue comme une anomalie.

LINUX_THREADS : Les listages 4.2 4.3 et 4.4 rapportent les lignes qui diffèrent entre les deux temps d'analyse et qui sont pertinentes à l'analyse. Les listages 4.2 et 4.3 rapportent principalement des résultats normalement attendus de toutes applications Android lors de leur exécution. Il s'agit des fils d'exécution que lance la plateforme à l'exécution d'une application. Le listage 4.4 quant à lui dénote une anomalie. En effet, les valeurs 0 dans les champs *uid_cred*, *gid_cred* et *euid_cred* démontrent que le nouveau processus *towelroot-dec* détient des fils d'exécution sous l'utilisateur *root*.

```
1  Offset  NameProc  TGID  ThreadPid  thread_offset  Addr_limit  uid  gid  euid
       ↪ ThreadName
2  [...]
3  0xec3ee800  com.example.dex  2849  2849  0xec3ee800  0x00000058  10087  10087  10087
       ↪ com.example.dex
4  0xec3ee800  com.example.dex  2849  2853  0xec3edc00  0x00000008  10087  10087  10087
       ↪ Heap thread poo
5  0xec3ee800  com.example.dex  2849  2854  0xec3ef800  0x00000283  10087  10087  10087
       ↪ Heap thread poo
6  0xec3ee800  com.example.dex  2849  2855  0xec826800  0xe02cf8d0  10087  10087  10087
       ↪ Heap thread poo
7  0xec3ee800  com.example.dex  2849  2861  0xebc5cc00  0x12c0cea0  10087  10087  10087
       ↪ Signal Catcher
8  0xec3ee800  com.example.dex  2849  2862  0xed0bf000  0x0efe0ada  10087  10087  10087
       ↪ JDWP
9  0xec3ee800  com.example.dex  2849  2863  0xecd1dc00  0xf8d868ad  10087  10087  10087
       ↪ ReferenceQueueD
10 0xec3ee800  com.example.dex  2849  2864  0xecd1d800  0x465a4651  10087  10087  10087
       ↪ FinalizerDaemon
11 0xec3ee800  com.example.dex  2849  2865  0xecd1d400  0x1c3970f4  10087  10087  10087
       ↪ FinalizerWatchd
12 0xec3ee800  com.example.dex  2849  2866  0xecd1d000  0x3038f24a  10087  10087  10087
       ↪ HeapTrimmerDaem
13 0xec3ee800  com.example.dex  2849  2867  0xecd1cc00  0x1c10e140  10087  10087  10087
       ↪ GCDaemon
14 0xec3ee800  com.example.dex  2849  2868  0xecd1c400  0x47f0004e  10087  10087  10087
       ↪ Binder_1
15 0xec3ee800  com.example.dex  2849  2869  0xebc5c800  0x0000008a  10087  10087  10087
       ↪ Binder_2
16 [...]
```

Listing 4.2 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_threads*.

```
1  Offset  NameProc  TGID  ThreadPid  thread_offset  Addr_limit  uid  gid  euid
       ↪ ThreadName
2  [...]
3  0xec3ee800  com.example.dex  2849  2877  0xecd1e800  0x1c48f643  10087  10087  10087
       ↪ Thread-268
4  0xec3ee800  com.example.dex  2849  2880  0xec896800  0x00310039  10087  10087  10087
       ↪ RenderThread
5  0xec3ee800  com.example.dex  2849  2883  0xec895400  0x12de0640  10087  10087  10087
       ↪ GL updater
6  0xec3ee800  com.example.dex  2849  2885  0xec895800  0x12c5d4c0  10087  10087  10087
       ↪ hwuiTask1
7  0xec3ee800  com.example.dex  2849  2886  0xec895c00  0x13e2cb20  10087  10087  10087
       ↪ hwuiTask2
8  0xec3ee800  com.example.dex  2849  2888  0xeb78c000  0x00001136  10087  10087  10087
       ↪ Binder_3
9  0xec3ee800  com.example.dex  2849  3441  0xebb4a400  0xf8d03a10  10087  10087  10087
       ↪ AsyncTask #1
10 0xec3ee800  com.example.dex  2849  3448  0xec914400  0x6841200c  10087  10087  10087
       ↪ .ProcessManager
11 0xec3ee800  com.example.dex  2849  3469  0xec914c00  0x000080e0  10087  10087  10087
       ↪ RenderThread
12 0xec3ee800  com.example.dex  2849  3472  0xecdbb400  0x00333ca8  10087  10087  10087
       ↪ RenderThread
13 0xec3ee800  com.example.dex  2849  3473  0xecdb8800  0x7c7af44f  10087  10087  10087
       ↪ RenderThread
14 0xec3ee800  com.example.dex  2849  3474  0xecdb9c00  0x1c3970e3  10087  10087  10087
       ↪ RenderThread
15 0xec3ee800  com.example.dex  2849  3475  0xeee49000  0x00000000  10087  10087  10087
       ↪ RenderThread
16 [...]
```

Listing 4.3 – (Suite...) Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_threads*.

```
1  Offset  NameProc  TGID  ThreadPid  thread_offset  Addr_limit  uid  gid  euid  ThreadName
2  0xec8fdc00  towelroot-dec  3451  3451  0xec8fdc00  0xc000f8dc  0  0  0  towelroot-dec
3  0xec8fdc00  towelroot-dec  3451  3452  0xec8ffc00  0x45560a01  0  0  0  towelroot-dec
4  0xec8fdc00  towelroot-dec  3451  3453  0xec8fe000  0x4680b091  0  0  0  towelroot-dec
5  0xec8fdc00  towelroot-dec  3451  3454  0xec8fc000  0xf8d00124  0  0  0  towelroot-dec
6  0xec8fdc00  towelroot-dec  3451  3455  0xec917000  0xf8d9e7bd  0  0  0  towelroot-dec
7  0xec8fdc00  towelroot-dec  3451  3456  0xec8ff800  0x0066f2c7  0  0  0  towelroot-dec
8  0xec8fdc00  towelroot-dec  3451  3457  0xec8ff400  0x000cf850  0  0  0  towelroot-dec
9  0xec8fdc00  towelroot-dec  3451  3458  0xec8fc400  0x1cacf64c  0  0  0  towelroot-dec
10 0xec8fdc00  towelroot-dec  3451  3459  0xec4b1800  0x9e9fde6c  0  0  0  towelroot-dec
11 0xec8fdc00  towelroot-dec  3451  0001  0xec4b1c00  0xbc0c2b78  0  0  0  towelroot-dec
12 0xec917c00  towelroot-dec  3465  3465  0xec917c00  0xe1b4f8d9  0  0  0  towelroot-dec
```

Listing 4.4 – (Suite...) Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_threads*.

LINUX_PSAUX : La comparaison réalisée a permis d'extraire le résultat présenté au listage 4.5.

```
PID     UID     GID     Arguments
3451    0       0       /data/data/com.example.dex/app_decrypted/towelroot-dec "/data/data/com.
    ↪ example.dex/app_decrypted/busybox-dec nc -l -p 5550 -e /system/bin/sh" &
```

Listing 4.5 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_psaux*.

Ce résultat présente la commande lancée par l'exploit. Cette commande lance un serveur *netcat* qui écoute sur le port 5550 dans le but de rediriger les commandes reçues et les sorties qui y sont associées à une invite de commande */system/bin/sh* sur le téléphone.

LINUX_ELFS : Le listage 4.6 rapporte les entrées pertinentes ajoutées par le lancement du comportement malicieux.

```
Pid       Name                Start       End         Elf Path
--------- ------------------- ----------- ----------- -------------------------
3451 towelroot-dec     0xb6efb000 0xb6efe030 /system/lib/libnetd_client.so
3451 towelroot-dec     0xb6f20000 0xb6f827c8 /system/lib/libc.so
3451 towelroot-dec     0xb6f83000 0xb6f9b0b0 /system/lib/libm.so
3451 towelroot-dec     0xb6f9c000 0xb6f9f00c /system/lib/libstdc++.so
3451 towelroot-dec     0xb6fb6000 0xb6fbb864 /data/data/com.example.dex/app_decrypted/
    ↪ towelroot-dec
3465 towelroot-dec     0xb6fb6000 0xb6fbb864 /data/data/com.example.dex/app_decrypted/
    ↪ towelroot-dec
```

Listing 4.6 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du cheval de Troie tel que rapporté par le module *linux_elfs*.

Le listage illustre, à nouveau, la présence du processus *towelroot-dec* de même que les fichiers ELF ayant été chargés par son lancement. L'observation de cette sortie permet de conclure que l'exécutable a été lancé à partir d'un dossier réservé à l'application *com.example.dex*, soit le maliciel à l'étude. Il est intéressant de noter que le chargement du fichier ELF */system/lib/-libnetd_client.so* constitue généralement un appel à une fonctionnalité réseau du SE. Dans le cas de *towelroot* [120], l'exploit utilise des *sockets* afin de mener son attaque ce qui explique le chargement de cette bibliothèque logicielle native.

LINUX_ARP : Le listage 4.7 présente les lignes qui diffèrent entre des deux moments d'analyse.

```
192.168.0.21 at 00:e6:1f:70:0f:f5 on wlan0
```

Listing 4.7 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_arp*.

La route menant à l'adresse IP 192.168.0.21 avec l'adresse physique 00:e6:1f:70:0f:f5 via l'interface wlan0 correspond à la route établie pour connecter le client distant à l'appareil infecté par le cheval de Troie.

LINUX_FIND_FILE : Le listage 4.8 présente les éléments pertinents à l'analyse et chargés en mémoire pendant l'exécution du comportement malicieux.

```
#Inode    Inode      Chemin d'acces
------ ---------- ---------
[...]
603441 0xed318518 /data/data/com.example.dex/app_decrypted
---    0x00000000 /data/data/com.example.dex/app_decrypted/busybox-dec nc -l -p 5550 -e
603520 0xed3cbe20 /data/data/com.example.dex/app_decrypted/towelroot-dec
---    0x00000000 /data/data/com.example.dex/app_decrypted/decrypted.bad
---    0x00000000 /data/data/com.example.dex/app_decrypted/busybox-dec
603439 0xed40ce38 /data/data/com.example.dex/app_outdex
603521 0xed3cbbd8 /data/data/com.example.dex/app_outdex/decrypted.dex
603437 0xed410088 /data/data/com.example.dex/app_malicious
603444 0xed2d0088 /data/data/com.example.dex/app_malicious/towelroot
603443 0xed3c19a0 /data/data/com.example.dex/app_malicious/payload.jar
603442 0xed3199a0 /data/data/com.example.dex/app_malicious/busyboxenc
[...]
```

Listing 4.8 – Résultats obtenus par l'analyse des différences entre les captures pré et post exécution du maliciel tel que rapporté par le module *linux_find_file*.

Dans ce listage, le *Inode Number* correspond à l'identifiant unique de la structure de données comportant le fichier dans la mémoire vive du système, le champ *Inode* à l'adresse où celui-ci se trouve et le champ *File Path* au chemin d'accès du fichier sur le système au moment du chargement.

L'utilisation du module *linux_find_file* avec l'option « *-i <Addr>* » permet d'extraire un fichier situé à l'adresse *<Addr>*, lorsque celle-ci n'est pas nulle ($0x00$) [1]. Avec cette option et en utilisant les valeurs d'adresse $0xed3cbbd8$ et $0xed3cbe20$, il est possible d'extraire les fichiers *decrypted.dex* et *towelroot-dec* respectivement. Le premier correspond au fichier de code chargé dynamiquement par le maliciel et le second à l'exploit *towelroot*. En mémoire, ces fichiers sont tous les deux décryptés. Il a été possible de confirmer que le fichier récupéré est identique à l'originale en comparant le résultat de leur somme de contrôle MD5 qui s'est révélé identique. En revanche, *busybox-dec* n'a pas été maintenu en mémoire vive dans un format décrypté et est, ainsi, irrécupérable par cette méthode. Ce résultat démontre bien le caractère imprévisible et sensible de l'AMV. Son contenu peut changer sans préavis et altérer la capture obtenue.

4.5 Discussion

L'utilisation de *Volatility* a permis de décomposer le contenu brut des captures de mémoire vive en différentes structures internes du noyau Linux du SE Android. L'utilisation de l'analyse différentielle a permis d'établir le delta entre t_{pre} et t_{post} afin de relever uniquement les changements d'intérêt dans ces structures étant survenus pendant la durée de l'exécution de l'application. Finalement, le triage final de ce différentiel par un analyste a permis d'identifier

1. La valeur nulle indique que le contenu du fichier n'est plus présent en mémoire.

	Comportements	Module(s)	Détection
1.	Obfuscation par le chiffrement des ressources	LINUX_ELFS LINUX_FIND_FILE	Partielle
2.	Chargement dynamique de code externe	LINUX_FIND_FILE	Détecté
3.	Suppression des fichiers intermédiaires	LINUX_FIND_FILE	Partielle
4.	Exécution de ressources binaires externes	LINUX_PSTREE LINUX_THREADS LINUX_PSAUX LINUX_ELFS	Détecté
5.	Élévation de privilèges	LINUX_PSTREE LINUX_THREADS LINUX_PSAUX	Détecté
6.	Connexion à un serveur de commande et contrôle	LINUX_PSAUX LINUX_ARP LINUX_FIND_FILE	Partielle

TABLEAU 4.1 – Comportements malicieux détectés par l'ADMV.

les comportements qui sont liés à l'exécution de l'application par opposition à ceux qui proviennent du fonctionnement normal du SE.

L'approche par l'ADMV a permis d'extraire des marqueurs révélant la présence du maliciel sur la plateforme. Le tableau 4.1 présente les stratégies employées par le maliciel, les greffons ayant permis de détecter le comportement décrit et si les comportements ont pu être détectés.

L'obfuscation par le chiffrement des ressources a pu être en partie détectée. En effet, il a été possible de détecter que des ressources n'appartenant pas à l'application d'origine ont été chargées par celle-ci au moment de l'exécution. Il a été également possible de récupérer certains de ces fichiers depuis la mémoire vive. Les ressources chiffrées n'ont pu être détectées avant d'être chargées par l'application et l'action du déchiffrement lui-même n'est pas captée par l'ADMV.

Le chargement dynamique de code externe à l'application a été détecté. Le code chargé dynamiquement a pu être détecté et récupéré en mémoire vive. Il a également été possible de valider que le contenu du code récupéré correspond au code d'origine.

La suppression des fichiers utilisés pour lancer le comportement malicieux a pu être détectée en partie. Le contenu du fichier déchiffré *busybox-dec* utilisé pour déployer le cheval de Troie n'a pu être récupéré. Cependant, le descripteur du fichier comportant notamment son nom a tout de même pu être relevé. Cela peut être suffisant pour mener à une analyse plus en profondeur de l'application.

L'élévation de privilèges a été détectée dans le contexte de l'application qui lance des opérations avec l'utilisateur *root*.

La connexion à un serveur de commande et contrôle a également été partiellement détectée. Les connexions et les commandes utilisées pour lancer le maliciel ont été clairement identifiées par l'ADMV. Il n'a été possible d'identifier les ports réseau ouverts par l'application qu'en partie. En effet, la commande « busybox−dec nc −l −p 5550 −e /system/bin/sh & » identifiée par le greffon LINUX_PSAUX comporte le numéro de port utilisé, mais ne peut constituer un marqueur fiable pour détecter des communications réseau. Un maliciel pourrait procéder autrement pour ouvrir une connexion distante avec un serveur de commande et contrôle. Dans un contexte où une application malicieuse dissimule ses communications dangereuses au travers de communications légitimes, les ports ouverts sur un appareil et les connexions avec les postes distants qui y sont rattachés faciliteraient le filtrage entre contenu légitime et illégitime. Il serait intéressant d'avoir une version fonctionnelle de LINUX_NETSTAT sur Android afin d'obtenir cette information et d'enrichir l'analyse des communications réseau sur l'appareil.

Également, cette méthode a été démontrée applicable sur appareils physiques en ne demandant qu'une modification mineure du noyau Linux. De ce fait, l'ADMV offre une alternative résiliente aux techniques de détection d'une machine virtuelle ou un émulateur.

4.6 Limites

L'étude menée dans le cadre de cette maîtrise comporte des limites. Tout d'abord, l'approche proposée a été évaluée sur une application conçue pour n'exprimer que le comportement malicieux étudié. Or, il est commun qu'une application exécute différents comportements (p. ex. lecture de média, affichage de contenu publicitaire, etc.) pendant son exécution. Dans un contexte d'AMV, cela peut avoir un impact négatif sur le contenu récupérable et sur la quantité de données inutiles à filtrer pendant la phase d'analyse. Il serait souhaitable d'approfondir la recherche afin d'identifier une approche qui permettrait d'obtenir successivement plusieurs captures de la mémoire vive. De cette façon, il serait possible de garder la trace du contenu qui y transige et de limiter les risques de perdre des marqueurs importants. Considérant le stockage requis pour chacune des captures obtenues dans les présents travaux qui sont égales à la capacité de la mémoire vive de l'appareil analysé (2 Go pour le Nexus 4) et le long temps d'acquisition nécessaire (environ 2 min), la technique est encore trop exigeante pour être utilisée sur des appareils destinés à des utilisateurs.

Une autre limitation des présents travaux est que l'application analysée a été conçue expressément pour l'étude. De ce fait, le comportement malicieux recherché était entièrement connu *a priori*. Dans une situation réelle, le concepteur du maliciel et l'analyste sont typiquement des entités différentes. Ce dernier doit avoir une compréhension du fonctionnement du SE afin d'être en mesure d'inférer s'il y a des activités suspectes à partir du résultat des différences entre les captures de la mémoire vive.

Dans l'optique d'améliorer l'efficacité de la technique, le devis a été créé en cherchant à limiter les comportements à risque d'interférer avec l'analyse, comme l'action d'autres applications. Cette décision méthodologie a pour effet de minimiser la corruption des artefacts d'intérêt en mémoire, mais également de minimiser la validité écologique de l'étude. Toutefois, dans un contexte d'un laboratoire d'analyse de maliciels, cette décision est justifiée, souhaitable et applicable.

4.7 Travaux futurs

Au final, l'AMV par l'utilisation d'un différentiel entre des captures aux temps t_{pre} et t_{post} est une approche complémentaire aux autres approches d'analyses dynamiques et statiques. Néanmoins, plusieurs pistes de recherche sont souhaitables pour l'expansion de ce domaine prometteur. Notamment, il est important que les travaux futurs s'intéressent à réduire le temps requis afin d'effectuer une capture intégrale de la mémoire vive sur l'appareil. Pour ce faire, il est nécessaire de développer des techniques d'acquisitions qui seraient capables, tout comme LiME, de faire une capture complète de la mémoire vive, mais également de cibler spécifiquement des processus d'intérêt. Cela aurait pour effet de réduire la quantité de mémoire à parcourir pour chaque capture. Une telle technique favoriserait des temps d'acquisition plus courts et limiterait le risque de manquer un comportement d'intérêt. De plus, cela permettrait d'effectuer des captures fréquentes afin de profiler les interactions d'une application avec son espace mémoire alloué pour détecter d'éventuelles anomalies en cas d'exploit. Il serait également intéressant d'automatiser le triage des résultats de l'ADMV afin d'isoler automatiquement les changements qui sont uniquement imputables à l'échantillon analysé plutôt que de nécessiter une inspection manuelle par un analyste.

Un autre axe de recherche pour ce domaine serait d'identifier une technique permettant d'injecter un module de capture de la mémoire vive dans la plateforme Android sans nécessiter de remplacer son noyau Linux au préalable. L'étude de Stüttgen et Cohen [111], dans laquelle un processus d'acquisition est injecté à même un LKM déjà présent en mémoire, a tenté de répondre à cette problématique. Toutefois, l'approche proposée dans leur étude nécessiterait tout de même de recompiler un noyau pour activer le support des LKM en plus de devoir être adaptée à Android. Dans le même but, l'approche par l'ajout à Android de modules de sécurité proposée par Heuser *et al.* [61] pourrait offrir des fonctionnalités permettant d'acquérir la mémoire vive de façon légitime. L'approche n'ayant pas reçu d'appui par Google dans le code source d'origine du SE, cette solution a reçu peu d'attention par les principaux fabricants d'appareils intelligents.

Résumé

Les résultats obtenus dans ce chapitre démontrent qu'il est possible d'utiliser les greffons de *Volatility* plus génériques (c.-à-d. qui ciblent le noyau Linux en général plutôt que de s'attarder aux structures spécifiques à Android) pour détecter des activités malicieuses sur un appareil physique, et ce, même si le maliciel tente d'éviter la détection. Il est donc pertinent de poursuivre la recherche dans le domaine de l'AMV afin d'élaborer de nouveaux outils d'analyse plus flexibles que ceux présentés dans les travaux précédents, plus efficaces que ceux utilisés dans l'étude de cas et plus spécialisés sur le SE Android afin d'obtenir des résultats plus détaillés. La proposition de l'ADMV dans un contexte d'analyse de maliciels est un pas en ce sens.