Compte-rendu du challenge SSTIC 2010

Auteur : Frédéric Guihéry

Contact : frederic.guihery@amossys.fr

Outils utilisés : hexedit, ghex2, python, scalpel, foremost, strings, file, SDK Android, GnuPG, pgpdump, dd, dexdump

1) Analyse du fichier *concours_sstic_2010*

```
$ wget http://www.sstic.org/media/SSTIC2010/concours_sstic_2010

$ file concours_sstic_2010
concours_sstic_2010: 7-zip archive data, version 0.3

$ 7zr e concours_sstic_2010

7-Zip (A) 9.04 beta  Copyright (c) 1999-2009 Igor Pavlov  2009-05-30
p7zip Version 9.04 (locale=fr_FR.utf8,Utf16=on,HugeFiles=on,1 CPU)

Processing archive: concours_sstic_2010

Extracting  challv2

Everything is Ok

Size:       100663296
Compressed: 23667438

$ file challv2
challv2: data
```

Nous voilà maintenant face au dump mémoire de l'Android. Comme le challenge est organisé par l'ANSSI, testons :

```
$ strings challv2 | grep -i anssi
...
com.anssi.secret.apk
com.anssi.textviewer.apk
...
```

Intéressant puisque, sous Android, les applications Java sont présentes sous forme d'archives APK (équivalent des archives JAR). Il s'agit donc maintenant de récupérer le contenu de ces deux archives pour le moins suspectes. Regardons du côté des outils de forensics.

2) Première tentative d'extraction des applications Java

```
$ foremost -o output_forensic -t all -i challv2
Processing: challv2
...

$ ls output_forensic
```

```
audit.txt  gif  htm  jpg  png  zip
```

Les archives APK étant simplement zippées, regardons ce que nous a trouvé foremost :

```
$ ls output_forensic/zip
00015304.zip   00063435.zip   00067531.zip   00079252.zip   00084204.zip   00091712.zip
00094636.zip  00098168.zip  00115876.zip  00144584.zip  00190464.zip
00016024.zip   00066120.zip   00067896.zip   00079408.zip   00085586.zip   00093472.zip
00095619.zip  00105200.zip  00122897.zip  00145337.zip
00028744.zip   00066460.zip   00068532.zip   00079828.zip   00088995.zip   00093698.zip
00097418.zip  00105464.zip  00123600.zip  00189264.zip
00061880.zip   00066836.zip   00070065.zip   00080309.zip   00091056.zip   00094096.zip
00097648.zip  00105976.zip  00131077.zip  00190104.zip

$ for i in *.zip; do unzip -d zip_$i $i ; done

$ grep -nir "com.anssi" .
Fichier binaire ./00105976.zip concordant
Fichier binaire ./00098168.zip concordant
Fichier binaire ./00095619.zip concordant
Fichier binaire ./00091712.zip concordant
Fichier binaire ./00131077.zip concordant
Fichier binaire ./00084204.zip concordant
Fichier binaire ./00097648.zip concordant
Fichier binaire ./zip_00123600.zip/classes.dex concordant
```

Une rapide analyse dans les archives décompressées montre qu'il manque beaucoup de fichiers. En particulier, l'archive 00028744.zip référence un fichier suspect, lib/armeabi/libhello-jni.so, qu'elle ne contient pas :

```
$ cat zip_00028744.zip/META-INF/CERT.SF
Signature-Version: 1.0
...
Name: lib/armeabi/libhello-jni.so
SHA1-Digest: q2NQjkOdcSsDp2WiV2PRxwnRMfk=
```

A ce moment là, on se souvient que le challenge repose sur un dump de mémoire physique, et qu'un mécanisme de pagination est probablement en place. Ceci explique pourquoi les archives APK sont incomplètes ; leur mappage en mémoire physique est réalisé sur des pages mémoire dispersées...
Le téléchargement du SDK d'Android nous donne alors plus d'informations sur l'architecture et l'OS sous-jacent.

```
$ ./emulator -avd google
$ ./adb shell
# cat /proc/cpuinfo
Processor    : ARM926EJ-S rev 5 (v5l)
...

# cat /proc/version
Linux version 2.6.29-00255-g7ca5167 (digit@digit.mtv.corp.google.com) (gcc version 4.4.0
(GCC) ) #9 Tue Dec 1 16:12:35 PST 2009
```

Nous voilà donc sur OS bien connu, et sur une architecture qui l'est un peu moins. Voyons maintenant voir comment retrouver les pages virtuelles des différents processus à partir de ce dump de pages physiques.

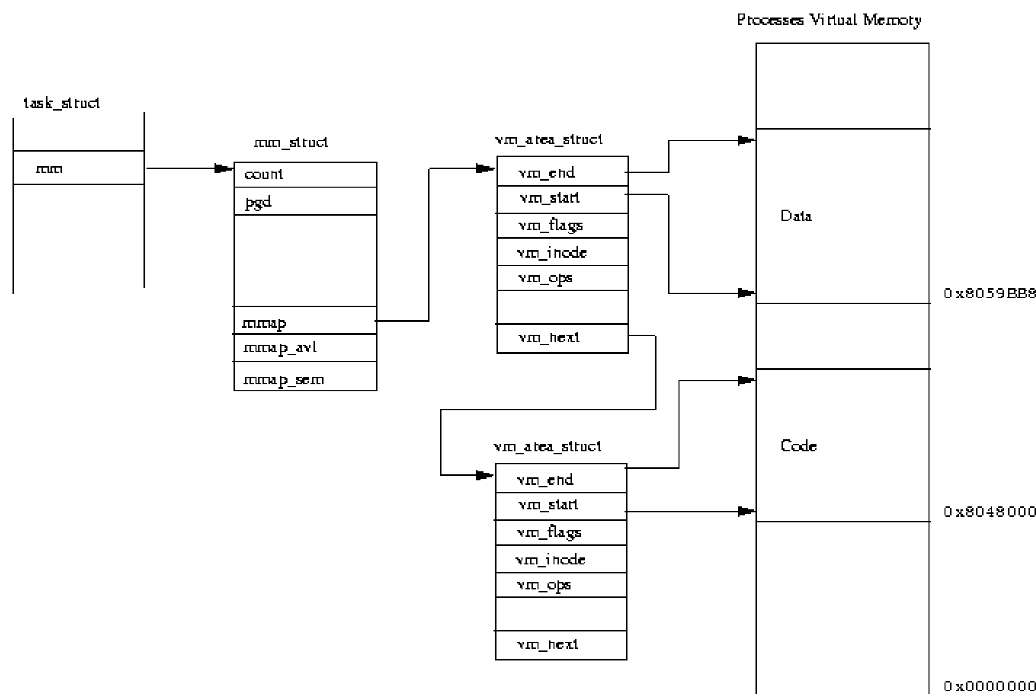
3) Reconstruction des espaces mémoire des processus

En parcourant les outils de forensics dédiés à la recherche de structures noyau (Draugr, Volatility, Idetect), on s'aperçoit qu'aucun ne convient réellement. Il nous faut donc re-coder cela à la main.

Première difficulté : trouver un point d'entrée dans la liste doublement chainée des task_struct. Quand on a accès au système de fichiers, il suffit de lire le fichier /boot/System.map pour trouver le symbole exporté de la structure init_thread_union du processus init. Celui-ci étant un bon point d'entrée pour retrouver la structure task_struct d'init. Malheureusement, le système de fichiers n'est pas fourni.

En regardant de plus près la structure task_struct[1], à coup de hexedit, et par recoupements successifs, on peut tout de même retrouver le point d'entrée de la task_struct du processus init : **0xC02A0F80** !

A partir de là, il suffit de se balader dans les structures du noyau (task_struct -> mm -> vma_struct[]) pour obtenir le mappage des zones mémoires dans l'espace mémoire virtuel des processus.



Source : http://tldp.org/LDP/tlk/kernel/processes.html

A partir de ce mappage, on obtient l'ensemble des pages virtuelles : il suffit de prendre la base d'une VMA et d'incrémenter par saut de 4k pour trouver l'adresse virtuelle de chaque page. On s'arrête lorsqu'on arrive sur le sommet de la VMA. On obtient également l'adresse du PGD dans la structure mm.

La traduction des adresses virtuelles en adresses physiques se fait en s'aidant du tableau suivant, issu des spécifications ARM9 :

1 http://lxr.linux.no/#linux+v2.6.29/include/linux/sched.h#L1114

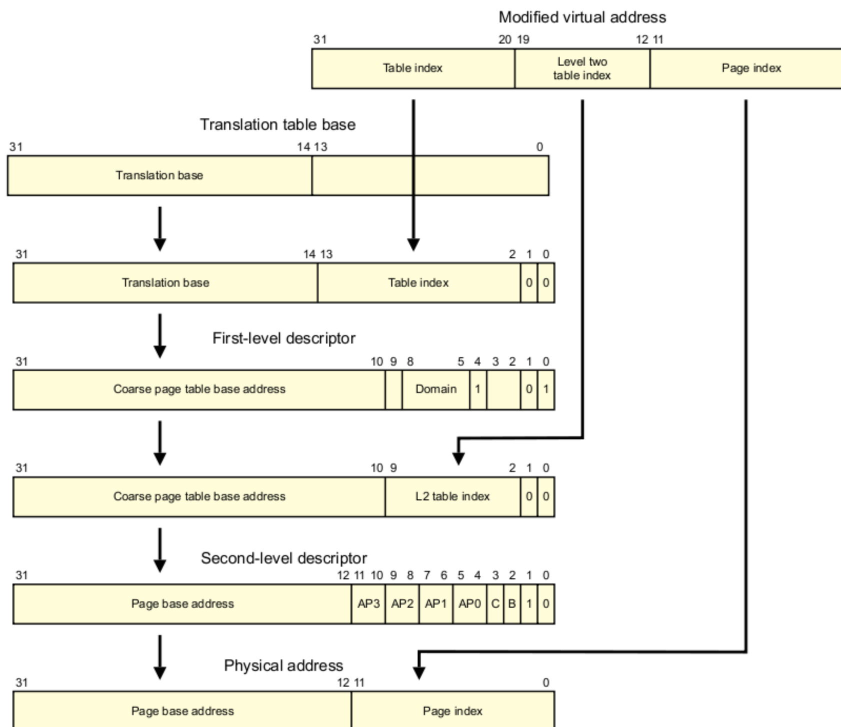


Figure 3-11 Small page translation from a coarse page table

Source : http://infocenter.arm.com/help/topic/com.arm.doc.ddi0198e/DDI0198E_arm926ejs_r0p5_trm.pdf

En réitérant ceci pour chaque processus de la liste chainée, on retrouve ainsi l'ensemble des couples (@page physique, @page virtuelle) de toutes les applications en cour d'exécution au moment du dump mémoire. L'étape suivante consiste à ré-assembler les espaces mémoire des processus en concaténant les pages physiques (ici, dd est notre ami).

Le script fourni en annexe automatise entièrement la tâche.

Lors de son exécution, nous identifions clairement les deux processus du challenge :

```
task i= 40  addr= 63111616   task name= nssi.textviewer
mm_struct= 0x3a776a0L
n_vma= 157

task i= 41  addr= 89296320   task name= om.anssi.secret
mm_struct= 0x3a77220L
n_vma= 164
```

4) Seconde tentative d'extraction des applications Java

```
$ foremost -o output_forensic2 -t all -i challv2
$ cd output_forensic2/zip/
$ for i in *.zip; do unzip -d zip_$i $i ; done
```

Cette fois-ci, on retrouve l'ensemble des fichiers des applications com.anssi.secret et com.anssi.textviewer. Deux fichiers attirent l'attention :

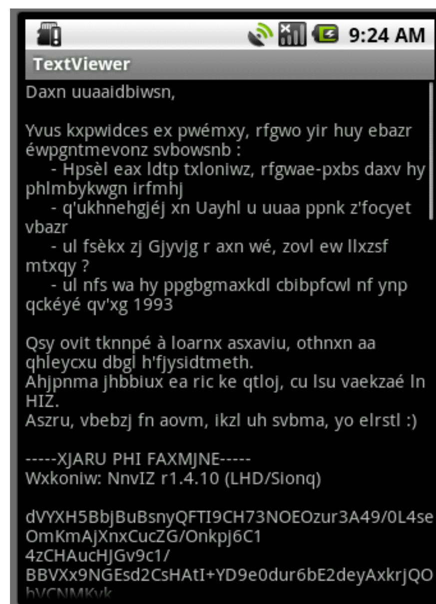
```
$ lz -R com.anssi.secret
...
libhello-jni.so
...
$ lz -R com.anssi.textviewer
...
chiffre.txt
...

$ file libhelleo-jni.so
secret/lib/armeabi/libhello-jni.so: ELF 32-bit LSB shared object, ARM, version 1 (SYSV),
dynamically linked, stripped
```

« chiffre.txt » s'avère être un fichier texte sur lequel un code de Vigénère a été passé.
Après avoir reconstruit les archives APK, nous pouvons les lancer dan l'émulateur Android :

```
./adb install com.anssi.secret.apk
./adb install com.anssi.textviewer.apk
```

L'exécution de Textviewer nous affiche simplement le fichier texte chiffre.txt :



L'exécution de l'application Secret nous demande plusieurs informations :

Une analyse du bytecode Java de Secret avec l'outil dexdump montre, qu'une fois les bonnes valeurs rentrées (coordonnées GPS et mot de passe), un code binaire est déchiffré avec RC4. Reste à trouver la clé RC4...
En précisant des coordonnées GPS quelconques, l'application plante : la sortie d'erreur nous indique que la bibliothèque JNI, appelée lorsque l'on clique sur « FINI », est corrompue. Une analyse de cette bibliothèque montre en effet qu'une page mémoire est manquante...

6) Réparation de la bibliothèque JNI

En regardant de plus près la résolution des pages virtuelles en pages physique, on s'aperçoit qu'une des pages mémoire contenant la bibliothèque JNI n'est plus présente en mémoire : le descripteur PTE indique une faute de page. Arf...

Avec un peu de chance, cette page, bien que marquée en faute, est peut-être toujours présente en mémoire. Une recherche dans le dump mémoire avec quelques motifs bien choisis confirme cette hypothèse. Les motifs choisis sont relatifs à la structure de donnée « ELF Sections » : cette structure est normalement présente (bien que optionnelle) dans chaque binaire ELF. Or, dans la bibliothèque JNI récupérée, cette structure n'apparait pas.

Ensuite, avec dd, il suffit de mettre bout à bout les pages physiques pour reconstruire l'ensemble de la bibliothèque JNI.

7) Déchiffrement du Vigénère

La clé Vigénère se trouve par déduction. En effet, une sous partie du message codé contient quelque chose qui ressemble à un entête PGP :

```
-----CXZES JPW PVUEEH ENF BMHVG-----
Ayazipg: ZjzJP c1.4.10 (GON/Eesog)
```

Avec « BEGIN PGP PUBLIC KEY BLOCK » comme message en clair, il suffit d'appliquer l'inverse de Vigénère sur ces quelques lettres pour retrouver la clé : **bttwfujha**.

Ensuite, en utilisant l'outil (http://sharkysoft.com/misc/vigenere/), on obtient le message suivant :

```
Cher participant,

Pour retrouver le trésor, rends toi aux lieux énigmatiques suivants :
        - Après les rump sessions, rendez-vous chez ce galliforme breton
        - l'abandonnée de Naxos y part pour d'autres cieux
        - le frère de Marvin y est né, sous la grosse table ?
        - le nez de ce gigantesque capitaine ne fut libéré qu'en 1993

Une fois arrivé à chaque endroit, valide ta position dans l'application.
Rajoute ensuite le mot de passe, il est chiffré en GPG.
Enfin, valide le tout, pour la suite, tu verras  :)

-----BEGIN PGP MESSAGE-----
Version: GnuPG v1.4.10 (GNU/Linux)

hQEOA5BaqIyWyerQEAP9GC73TFXOyby3E49/0G4yvHmJtHnStoVubGN/Siqgc6C1
...
uqy7QId+gb35oL4z6SotvOAcMNQj1cCnKNgQiYsnEhTsuv7JZjBqlm16DEzv
=vexd
-----END PGP MESSAGE-----

Je t'ai remis ma clef publique si tu veux me contacter.

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.10 (GNU/Linux)

mQGiBEug+8kRBAC1eP91t96pb/Ja59uVmbz4FuTag7eEFcoWTPUd5mqU9L0OsZH2
...
HodDDO5GUEA7AKDhvaeXaYoyjLLftlQY4WDssi91vgCfWWNio0oCfm0eTgCNc/im
ZBJoifI=
=8IMk
-----END PGP PUBLIC KEY BLOCK-----
```

Ce message contient 4 énigmes, un message chiffré (un mot de passe) et une clé publique ElGamal.

8) Résolution des énigmes

Une fois décodée, le fichier chiffre.txt nous propose plusieurs énigmes. La réponse à chaque énigme est en fait un lieu géographique, à partir duquel est obtenue une coordonnée GPS.

```
Pour retrouver le trésor, rends toi aux lieux énigmatiques suivants :
        - Après les rump sessions, rendez-vous chez ce galliforme breton
        - l'abandonnée de Naxos y part pour d'autres cieux
        - le frère de Marvin y est né, sous la grosse table ?
        - le nez de ce gigantesque capitaine ne fut libéré qu'en 1993
```

Réponse 1 : le restaurant Coq Gadby à Rennes : 48.1N 1.7W.

Réponse 2 : Kourou, d'où s'envole Ariane : 5.2N 52.8W.

Réponse 3 : le GooglePlex près de Palo Alto : 37.5N 122.1W.

Réponse 4 : la voie d'escalade The Nose, sur la montage El Capitan aux Etats-Unis, dont la première ascension libre a été réalisée en 1993 (merci fréro !) : 38N 119.6W.

Le lien entre le siège de Google et Marvin est quand même bien tiré par les cheveux : Marvin => Marvin l'Android => Android, l'OS de Google => qui a comme frère ChromeOS => qui a été "fabriqué" au siège de Google à Palo Alto => « sous » BigTable, qui est le SGBD de Google.

Dans un premier temps, n'arrivant pas à résoudre cette troisième énigme, une rétro-ingénierie du challenge s'imposait. Étant donné qu'avec les trois autres coordonnées GPS, on obtenait un texte presque cohérent dans l'application com.anssi.secret : « com.##ssi.se##et/RC4 », il était possible, par déduction (et avec un peu de brute-force), de retrouver une coordonnée GPS qui donne les bonnes lettres. En réalité plusieurs coordonnées GPS fonctionnent.

9) Cassage de la clé ElGamal

L'analyse du message chiffré montre que la clé de session est chiffrée avec deux clés publiques :

```
$ pgpdump message.txt
Old: Public-Key Encrypted Session Key Packet(tag 1)(270 bytes)
	New version(3)
	Key ID - 0x905AA88C96C9EAD0
	Pub alg - ElGamal Encrypt-Only(pub 16)
	ElGamal g^k mod p(1021 bits) - ...
	ElGamal m * y^k mod p(1021 bits) - ...
		-> m = sym alg(1 byte) + checksum(2 bytes) + PKCS-1 block type 02
Old: Public-Key Encrypted Session Key Packet(tag 1)(140 bytes)
	New version(3)
	Key ID - 0x2A9C6105E1F67BBD
	Pub alg - RSA Encrypt or Sign(pub 1)
	RSA m^e mod n(1021 bits) - ...
		-> m = sym alg(1 byte) + checksum(2 bytes) + PKCS-1 block type 02
New: Symmetrically Encrypted Data Packet(tag 9)(60 bytes)
	Encrypted data [sym alg is specified in pub-key encrypted session key]
```

Ne trouvant aucun indice sur une quelconque clé privée RSA ou El Gamal dans le dump mémoire, nous pouvons en déduire qu'il s'agit d'un challenge crypto, dont le but est de retrouver la clé privée associée à la clé publique ElGamal.

Et en effet... Le p-1 (p étant le modulo ElGamal) se factorise en petits facteurs en utilisant l'algorithme de Pollard et de Lenstra (merci à Frédéric Rémi pour l'idée !). Ensuite, l'algorithme Pohlig Hellman, utilisé pour le calcul du logarithme discret, et qui présuppose que p-1 est un nombre lisse, permet de retrouver la clé privée ElGamal (la valeur X).

En utilisant le programme situé ici (http://www.alpertron.com.ar/DILOG.HTM), nous obtenons la clé privée :

```
Exp (clé privée)=
8216000284630378878087764573899538969348205656695755958036937166108823945599870897923017
4270492184887093527756473793037846785162980029537236246269728347946278173732394323079906
05905765645459702616102054707

Durée du calcul = 1jour 18heures 27 minutes 42 secondes
```

Pour l'anecdote, le calcul a été lancé le temps du week-end de la Pentecôte sur un PC du boulot. La réponse était affichée le mardi au petit matin. Le programme, une applet Java, était lancé dans un navigateur situé dans une VM Virtualbox... Bref, une implémentation en code natif serait clairement plus rapide.

Afin de créer une bi-clé ElGamal au format GnuPG, nous modifions le code source de GnuPG pour forcer les paramètres ci-dessous lors de la génération d'une clé.

Le message peut alors être déchiffré. Nous obtenons ainsi le mot de passe de l'application Secret : **O7huQcYzHEPSq82m**.

9) Déchiffrement de l'application finale

On rentre les bonnes coordonnées GPS dans l'application com.anssi.secret :

```
$ telnet 127.0.0.1 5554
geo fix -1.7 48.1
geo fix -52.8 5.2
geo fix -122.1 37.5
geo fix -119.6 38
```

La bibliothèque JNI dérive alors une chaine de caractère : « com.anssi.secret/RC4 ». Il s'agit d'une classe Java, dont une méthode est appelée pour déchiffrer un code binaire situé dans la classe Secret.

La clé de déchiffrement RC4 est obtenue par une dérivation du mot de passe, également réalisée par la bibliothèque JNI. Au final, l'application com.anssi.secret génère un fichier nommé « binaire ».

```
$ file binaire
binaire: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), corrupted section header size

$ strings binaire
...
Bravo, le challenge est termin
 ! Le mail de validation est : %s
42849d74a8af53aa7a85fc4e956b2d84@sstic.org
```

Étant un fichier ELF pour architecture ARM, nous pouvons l'exécuter dans l'émulateur Android :

```
# cd /data/data/com.anssi.secret/files
# ls
binaire
# chmod 777 binaire
# ./binaire
Bravo, le challenge est termin� ! Le mail de validation est : 4284d974a8af53aa7a85fc4e956b2d84@sstic.org
```

A cet instant, ne voyant pas (et n'imaginant pas) une différence entre la sortie de la commande *strings* et la sortie lors de l'exécution du binaire, je choisis d'envoyer la première réponse. Nous sommes le mardi 25 mai, veille de la clôture du challenge. On m'informe un peu plus tard que ma réponse est erronée. En effet, lors de l'exécution du binaire, deux caractères dans l'adresse mail sont inversés... Ma seconde réponse, correcte cette fois-ci, arrive le 27. Hors délai.

Note: merci à mes collègues pour les discussions animées lors des pauses café ;)

Annexe

Script de reconstruction des espaces mémoire virtuels des processus Android.

```
#!/usr/bin/python

import commands

file = "../challv2"
init_task_addr = 2756480 # == C0 2A 0F 80
page_offset = 3221225472
tasks_offset = 448

def conv_val(v):
    return int(a[6:8] + a[4:6] + a[2:4] + a[0:2], 16)

def get_addr(a):
    return int(a[6:8] + a[4:6] + a[2:4] + a[0:2], 16)

def dump(fd, offset, length):
    try:
        fd.seek(offset)
        res = fd.read(length)
        return res.encode("hex")
    except IOError:
        return "00000000"

fd = open(file)

fd.seek(init_task_addr + tasks_offset)
next_task_tasks = fd.read(4)
next_task_tasks = get_addr(next_task_tasks.encode("hex")) - page_offset

pte_base = 0
i = 0
tmp = 0
while tmp != init_task_addr + tasks_offset:
    next_task_tasks = get_addr(dump(fd, next_task_tasks, 4)) - page_offset
    mm_struct = get_addr(dump(fd, next_task_tasks + 8, 4)) - page_offset

    tmp = next_task_tasks
    if 1:
        n_oseek = 0
        output_command = "dd if=/dev/zero of=/tmp/challenge_" + str(next_task_tasks) + " bs=4096 count=1"

        print "task i=", i, " addr=", next_task_tasks
        print "  mm_struct=", mm_struct

        i = i + 1

        # task name
        fd.seek(next_task_tasks + 69*4)
        task_name = fd.read(16)
        print "  task name=", task_name

        # PGD base
        pgd_base = get_addr(dump(fd, mm_struct + 9*4, 4)) - page_offset
        print "  pgd_base=", hex(pgd_base)
        if pgd_base < 0:
            continue

        # VMA
        n_vma = get_addr(dump(fd, mm_struct + 12*4, 4))
        print "  n_vma=", n_vma

        vma_struct = get_addr(dump(fd, mm_struct, 4)) - page_offset

        for j in range(0, n_vma):
            vma_struct = get_addr(dump(fd, vma_struct + 12, 4)) - page_offset

            if vma_struct > 0:
                vma_start = get_addr(dump(fd, vma_struct + 4, 4))
                vma_end = get_addr(dump(fd, vma_struct + 8, 4))
                n_pages = (vma_end - vma_start) / 4096
#                print "  vm_",j ,": start=", hex(vma_start), ", end=", hex(vma_end), ", n_pages=", n_pages
                for k in range(0, n_pages):
#                    print "    ", hex(vma_start+page_offset+4096*k)
                    pgd_offset = pgd_base + (((vma_start+4096*k) >> 20) << 2)
                    pmd_base_desc = get_addr(dump(fd, pgd_offset, 4))
#                    print "    ", k, " ", hex(pgd_offset), " ->", hex(pmd_base_desc)

                    # attention aux pages fault
                       if not (((pmd_base_desc & 0xf) == 0) or ((pmd_base_desc & 0xf) == 4) or ((pmd_base_desc &
0xf) == 8) or ((pmd_base_desc & 0xf) == 12)):
                        pmd_offset = ((pmd_base_desc>>10)<<10) + ((((vma_start+4096*k) & 0xFFFFF) >> 12) << 2)
                        pte_base_desc = get_addr(dump(fd, pmd_offset, 4))
#                    print "    ", k, " ", hex(pmd_offset), " ->", hex(pte_base_desc)

                    # attention aux pages fault
                       if not (((pte_base_desc & 0xf) == 0) or ((pte_base_desc & 0xf) == 4) or ((pte_base_desc &
0xf) == 8) or ((pte_base_desc & 0xf) == 12)):
                        pte_base_addr = (pte_base_desc >> 12) << 12
                        pte_base = (pte_base_desc >> 12)
```

```
#                        print "    pte_base =", pte_base

                         output_command = "dd if=" +file+ " of=/tmp/challenge_" +str(next_task_tasks)+ " bs=4096
count=1 skip=" +str(pte_base)+ " seek="+str(n_oseek)
                        commands.getstatusoutput(output_command)
                    else: # on met du vide dans la page correspondante
                          output_command = "dd if=/dev/zero of=/tmp/challenge_" +str(next_task_tasks)+ " bs=4096
count=1 seek="+str(n_oseek)
                        commands.getstatusoutput(output_command)

                    n_oseek = n_oseek + 1

fd.close()
```