

# Une solution du challenge SSTIC 2010

*Ou l'art de se retrouver en slip coton!*

Stéphane Duverger

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Recherche manuelle d'informations</b>	<b>2</b>
<b>3</b>	<b>Reconstruction de la mémoire virtuelle des tâches</b>	<b>5</b>
3.1	Retrouver une <i>task struct</i> . . . . .	5
3.2	Récupérer toutes les tâches . . . . .	6
3.3	Analyse des vmas . . . . .	7
<b>4</b>	<b>Analyse de TextViewer</b>	<b>11</b>
4.1	Désassemblage . . . . .	11
4.2	Déchiffrement du texte . . . . .	12
<b>5</b>	<b>Analyse de SecretJNI</b>	<b>13</b>
5.1	Désassemblage . . . . .	15
5.1.1	SecretJNI.smali . . . . .	15
5.1.2	RC4.smali . . . . .	17
<b>6</b>	<b>Analyse de libhello-jni.so</b>	<b>17</b>
6.1	Génération de signatures . . . . .	18
6.2	Analyse du code utile . . . . .	20
6.2.1	Identification de la fonction de hachage . . . . .	20
6.2.2	Reconnaissance des appels aux fonctions du NDK . . . . .	21
6.2.3	Analyse de <i>deriverclef</i> . . . . .	22
6.2.4	Récapitulatif . . . . .	24
<b>7</b>	<b>Résolution du challenge</b>	<b>24</b>
7.1	Intuition sur le nom de la classe recherchée . . . . .	24
7.2	Instrumentation de libhello-jni.so . . . . .	25
7.3	Génération du binaire correct . . . . .	27
7.4	C'est gagné ... par contre non! . . . . .	28
<b>8</b>	<b>Conclusion</b>	<b>29</b>

## 1 Introduction

Le challenge SSTIC édition 2010 consistait cette année à retrouver une adresse e-mail dans l'image de la mémoire physique d'un système Android. Un challenge ma foi d'un fort beau gabarit étant donné que le système Android est assez récent et que peu d'outils d'analyse existent à ce jour. Ci-dessous une liste d'outils *existants* ayant été utilisés :

- la suite logicielle Android (SDK, NDK)
- IDA (Win32 5.5)
- outils classiques Unix (strings, gdb, ld, ...)
- baksmali (désassembleur Dalvik)

Nous ne détaillerons pas les étapes d'installation et de configuration de la suite logicielle Android, ni d'un gdb 7.1 pour x86 et ARM, parce qu'elles sont tout simplement inintéressantes. Notre démarche de résolution a été la suivante :

- collecter un maximum d'informations à l'aide d'outils classiques
- prendre conscience que l'on se retrouve en slip
- développer les outils qui nous intéressent

## 2 Recherche manuelle d'informations

Une fois le fichier récupéré, nous commençons par détecter son format :

```
# file concours_sstic_2010
concours_sstic_2010: 7-zip archive data, version 0.3
```

Ca commence bien, nous ne disposons même pas de l'outil nécessaire à la décompression.

```
# sudo apt-get install p7zip
# 7zr e concours_sstic_2010
[...]
Extracting challv2
```

Une fois en possession de l'image de la mémoire physique décompressée, nous effectuons une recherche textuelle classique à l'aide de `strings`. On trouve énormément d'informations, notamment sur des applications lancées sur le système, la version du noyau linux utilisée ainsi que la board du périphérique Android afin de cibler un peu plus l'origine de cette image mémoire.

Les boards Goldfish sont celles utilisées pour la plateforme de l'émulateur, ainsi une recherche un peu plus orientée nous confirme qu'il s'agit bien d'une plateforme d'émulateur :

```
# strings challv2 | grep -i goldfish
<4>Machine: Goldfish
[...]
```

Une autre information utile, la version du noyau (en espérant que celle-ci n'ait pas été volontairement trafiquée) afin d'avoir une idée de la version d'Android utilisée et nous permettre de créer notre instance d'émulation pour commencer à travailler :

```
# strings challv2 | grep -i linux
Linux version 2.6.29-00255-g7ca5167 [...]
```

On sent très rapidement qu'il s'agit d'une image provenant d'une instance émulée d'un Android assez récent. Nous créons donc notre propre AVD Android 2.1 API 7 et décidons de générer une image de sa mémoire physique :

```
# emulator -avd challenge -qemu -monitor stdio
(qemu) pmemsave 0 100663296 "my_image"
```

En comparant les deux images, nous trouvons de nombreuses similitudes, du reste, suffisamment pour valider le fait que le challenge a bien été généré sous un émulateur. L'idée<sup>1</sup> nous vient subitement de générer un fichier comprenant les chaînes de caractères exclusivement présentes dans le dump mémoire du challenge :

```
#!/usr/bin/env python

standard = {}
f = open("strings_my_image")
for s in f:
    standard[s]=0
f.close()

out = open("unique", "w")
f = open("strings_challenge")
for s in f:
    if not standard.has_key(s):
        out.write(s)
f.close()
out.close()
```

A l'origine les fichiers de strings avaient une taille d'approximativement 8Mo, ce qui était assez lourd à analyser. Le fichier *unique* résultant ne fait plus que 691Ko, ce qui est largement acceptable pour une excursion manuelle.

Nous retrouvons tout un tas d'informations intéressantes. Tout d'abord des informations sur des applications tierces ayant été installées :

```
New package installed in /data/app/com.anssi.textviewer.apk
New package installed in /data/app/com.anssi.secret.apk
Start proc com.anssi.textviewer
Start proc com.anssi.secret for activity com.anssi.secret/.SecretJNI
Trying to load lib /data/data/com.anssi.secret/lib/libhello-jni.so
Added shared lib /data/data/com.anssi.secret/lib/libhello-jni.so
```

Puis un extrait de texte conservant sa ponctuation, mais pas vraiment intelligible :

```
Daxn uuaaidbiwsn,
Yvus kxpwidces ex pw
mxy, rfgwo yir huy ebazr
wpgntmevonz svbownb :
[...]
-----XJARU PHI FAXMJNE-----
Wxkoniw: NnvIZ r1.4.10 (LHD/Sionq)
dVYXH5BbjBuBsnyQFTI9CH73NOEOzur3A49/0L4seOmKmAjXnxCucZG/Onkpbj6C1
[...]
JPSuefYofQAsQH5ouSWfIN9tZPeqEM50kNQ+jZvAJrNJADvYWpop+8rChpFBHKLl
NBTZYbeRIwNVUdZCJnkLVpIBUpLClzXYR4UJJglJyd6MKfGuSeWKUX `Z
```

1. L'auteur de la solution a également rapidement pris conscience qu'il venait de réimplémenter "sort | uniq".

Et finalement, des informations provenant probablement de l'application *secret* :

```
Challenge SSTIC
477689b3cb25eba2b9d671cb4 [...] f9018f081df0fe3a2
7Bravo ! Va lancer le binaire pour voir si
a a march
Lcom/anSSI/secret/R$attr;
Lcom/anSSI/secret/R$id;
Lcom/anSSI/secret/R$layout;
Lcom/anSSI/secret/R$string;
Lcom/anSSI/secret/R;
Lcom/anSSI/secret/RC4;
Lcom/anSSI/secret/SecretJNI;
No GPS
RC4.java
SecretJNI.java
TableRow01
TextView02
TextView04
View01
[Landroid/widget/CheckBox;
binaire
abmV3c29mdCwgdHUgZXMGaW50ZXJkaXQgZGUgY2hhbGxlbmdlIHVvdXIgc29j
aWFsIGVuZ2luZWVyaW5nIGV4Y2Vzc2lmLg==
chiffre
clef
coincoin
```

Nous reconnaissons à l'œil nu un extrait d'une chaîne encodée en base64. Avant de pousser toute analyse, nous décidons d'essayer de la décoder :

```
>>> "aWFsIGVuZ2luZWVyaW5nIGV4Y2Vzc2lmLg==" .decode("base64")
'ial engineering excessif.'

>>> "abmV3c29mdCwgdHUgZX [...] Vzc2lmLg==" .decode("base64")
'i\x9b9\x95\xdd\xcd\xbd\x99\x99\x99\x99 [...] \xcd\xcd\xa5\x98\x98'

>>> "bmV3c29mdCwgdHUgZX [...] Vzc2lmLg==" .decode("base64")
'newsoft, tu es interdit de challenge pour social engineering excessif.'
```

Voilà une phrase qui réchauffe le cœur. Après cette brève analyse manuelle, nous sommes en possession des informations suivantes :

- il semblerait qu'au moins 2 applications aient été développées par l'ANSSI : *textviewer* et *secret*
- l'application *secret* semble utiliser une interface native *libhello-jni.so*
- l'application *secret* à l'air d'aimer la cryptographie et notamment RC4
- à ce stade d'analyse, il nous est impossible de savoir à quelle application appartient le texte chiffré et s'il est complet

L'idéal serait de disposer de ses applications afin de les analyser intégralement. Android étant basé sur un noyau Linux, nous décidons de développer un outil qui reconstituera la mémoire virtuelle de chaque tâche présente sur le système au moment de la génération de l'image. Ceci nous permettra d'isoler, de manière contiguë, les pages de mémoire physique associées à chaque application.

## 3 Reconstruction de la mémoire virtuelle des tâches

Rien ne nous garantit que l'allocateur de mémoire physique du noyau Linux a alloué la mémoire des tâches de manière contiguë. Notre but est donc de retrouver les structures noyaux décrivant les tâches et principalement leur environnement mémoire, afin de générer des fichiers correspondant à chacune de leurs vmas, qui "elles" sont des zones de mémoire contiguë.

### 3.1 Retrouver une *task struct*

Nos précédentes recherches nous ont permis de penser que le noyau utilisé était un noyau classique Android 2.1 pour Goldfish (2.6.29-00255-g7ca5167). Par défaut, ceux-ci ne supportent pas le chargement de modules. Nous décidons donc de recompiler un noyau ayant le support des modules afin d'en développer un minimaliste nous permettant de récolter différents offsets de champs de la *task struct* :

- *stack* pour récupérer le *thread info*
- *mm* pour récupérer le page directory et les vmas
- *comm* pour le nom de la tâche
- *tasks* pour la liste chaînée des tâches

```
static void tasks_info(void)
{
    struct thread_info *tinfo = current_thread_info();
    struct task_struct *task = tinfo->task;
    struct task_struct *ctask = task;
    unsigned int *ptr, nr;
    nr = 0;

    printk("offset from task to comm = %d\n"
           "current thread info 0x%p\n"
           "mm offset to task 0x%x\n"
           "pgd offset to mm 0x%x\n"
           "prev offset 0x%x next offset 0x%x from task\n"
           , (unsigned int)&ctask->comm[0] - (unsigned int)ctask
           , current_thread_info()
           , (unsigned int)&ctask->mm - (unsigned int)ctask
           , (unsigned int)&ctask->mm->pgd - (unsigned int)ctask->mm
           , (unsigned int)&ctask->tasks.prev - (unsigned int)ctask
           , (unsigned int)&ctask->tasks.next - (unsigned int)ctask );

    printk("U/K task stack pgd name thread_info\n");
    do{
        printk("%d 0x%p 0x%p 0x%p %s (@0x%x)\n",
               ctask->mm?1:0,
               ctask,
               ctask->stack,
               ctask->mm?ctask->mm->pgd:init_mm.pgd,
               ctask->comm, ctask->comm );
    } while ((ctask=next_task(ctask)) != task);
}
```

Notons que ces offsets sont dépendant des options de configuration du noyau (architecture, fonctionnalités, ...).

L'émulateur permet ensuite de facilement démarrer notre AVD avec un noyau personnalisé et de récupérer les logs kernels générés.

L'idée est à présent de trouver un bon point d'entrée dans l'image de la mémoire physique afin de commencer notre parcours de la liste des tâches. Nous décidons de rechercher le nom d'un thread kernel présent sur la plupart des

systèmes Linux : "swapper". La *task struct* stocke le nom des threads dans un tableau (*comm*) ayant une taille fixe et dont nous connaissons désormais l'offset par rapport au début de la *task struct*.

Si nous retrouvons cette chaîne dans la mémoire physique, nous pourrions commencer à parcourir la liste des tâches et dumper leurs vmas. Un simple éditeur hexadécimal nous indique que la chaîne se trouve à l'offset 0x2a1254 depuis le début de l'image de la mémoire physique. Nous savons que Linux mappe la mémoire physique en identity mapping, à *PAGE\_OFFSET* près. Les offsets dans le fichier représentant la mémoire physique correspondront ainsi à des adresses physiques, en ce qui concerne la mémoire noyau.

A partir de ce point quelques vérifications vont être nécessaires pour valider nos heuristiques. Nous récupérons, à partir du pointeur de *task struct*, l'adresse de la pile noyau de *swapper*, contenue dans le champ *stack*. Cette adresse de pile noyau correspond au début de la structure *thread info*, au sein de laquelle nous récupérons le champ *task*. Ce dernier doit être égal à notre pointeur de *task struct*. Si cette vérification est correcte, nous pouvons considérer que nous avons effectivement récupéré la *task struct* de *swapper*.

```
unsigned int find_swapper(unsigned int pmem_offset)
{
    unsigned int task = (pmem_offset - task_to_name);
    unsigned int thrd = dump.p32[(task+task_to_stack)/4] - page_offset;
    unsigned int thrd_task = dump.p32[(thrd+thrd_to_task)/4] - page_offset;

    if (task != thrd_task)
        return 0;

    return task;
}
```

## 3.2 Récupérer toutes les tâches

Une fois *swapper* trouvé, il ne reste plus qu'à parcourir la liste des tâches et dumper leurs vmas dans des fichiers distincts. Notons que les pages de mémoire physique non-présentes, ainsi que les *memory mapped devices* et toutes références à des adresses physiques situées au-delà de la taille de la RAM, sont remplacées par des pages remplies de zéro afin de conserver des offsets corrects dans les fichiers représentant les vmas.

```
void reload()
{
    unsigned int task, next;

    if (!(task=find_swapper(swapper_off)))
        fail("swapper not found");

    mkdir(base_dir);

    /* user memory dumps */
    next = task;
    do{
        dump_umem(next);
        next = next_task(next);
    }while (next != task);

    dump_orphans();
}
```

La mémoire noyau n'a pas été récupérée, afin de facilement indexer les pages

de mémoire physique non-référencées (*orphelines*). Le noyau Linux mappe par défaut toute la mémoire physique, ce qui ne laissait aucune chance à notre outil d'indexer les pages de mémoire physiques orphelines. Une solution plus élégante pourrait être trouvée afin de permettre le référencement "réel" de la mémoire utilisée par le noyau.

```
int dump_umem(unsigned int task)
{
    unsigned int mm, vmas, start, end, pgd;
    char        *name;
    char        tpath[128];
    char        fname[128];

    if (! (mm=dump.p32[(task+task_to_mm)/4]) )
        return 1;

    name = task_name(task);
    sprintf(tpath, base_dir"/%s", name);

    mkdir(tpath);

    printf("dump task mem: %s\n", name);

    mm -= page_offset;
    vmas = dump.p32[(mm+mm_to_vma)/4];
    pgd = dump.p32[(mm+mm_to_pgd)/4] - page_offset;

    while (vmas){
        vmas -= page_offset;
        start = dump.p32[(vmas+vma_to_start)/4];
        end   = dump.p32[(vmas+vma_to_end)/4];

        sprintf(fname, "%s/vma_0x%x_0x%x", tpath, start, end);
        if (! virt_to_phys_dump(pgd, fname, start, end))
            fail("dump vma [0x%x - 0x%x]", start, end);

        vmas = dump.p32[(vmas+vma_to_next)/4];
    }
    return 1;
}
```

Notons également que nous ne nous sommes pas attardés sur la représentation interne de la mémoire virtuelle du noyau Linux sous architecture ARM (pmd, dirty bit emulation, ...). Nous avons simplement implémenté le code nécessaire à la lecture des pde/pte telles qu'elles sont vues par un processeur ARM.

Le code de l'outil reconstruisant la mémoire des tâches est joint avec la présente solution.

### 3.3 Analyse des vmas

Une fois l'exécution terminée, nous nous retrouvons avec les répertoires suivants :

```

# ll memory/
total 636K
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 adbd
drwx----- 2 stf stf 12K 2010-05-19 13:23 com.android.mms
drwx----- 2 stf stf 12K 2010-05-19 13:23 com.svox.pico
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 debuggerd
drwx----- 2 stf stf 12K 2010-05-19 13:23 d.process.acore
drwx----- 2 stf stf 12K 2010-05-19 13:23 d.process.media
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 init
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 init.goldfish.s
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 installd
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 keystore
drwx----- 2 stf stf 12K 2010-05-19 13:23 m.android.email
drwx----- 2 stf stf 12K 2010-05-19 13:23 m.android.phone
drwx----- 2 stf stf 12K 2010-05-19 13:23 mediaserver
drwx----- 2 stf stf 12K 2010-05-19 13:23 ndroid.settings
drwx----- 2 stf stf 12K 2010-05-19 13:23 nssi.textviewer
drwx----- 2 stf stf 12K 2010-05-19 13:23 om.anssi.secret
drwx----- 2 stf stf 408K 2010-05-19 13:23 ___orphans__
drwx----- 2 stf stf 12K 2010-05-19 13:23 putmethod.latin
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 qemud
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 qemu-props
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 riid
drwx----- 2 stf stf 12K 2010-05-19 13:23 roid.alarmclock
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 servicemanager
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 sh
drwx----- 2 stf stf 20K 2010-05-19 13:23 system_server
drwx----- 2 stf stf 4.0K 2010-05-19 13:23 void
drwx----- 2 stf stf 12K 2010-05-19 13:23 zygote

```

Chaque répertoire contient la liste des vmas de la tâche. Le répertoire `___orphans__` contient la liste des pages de mémoire physique non-référencées. La granularité la plus fine (4Ko) a été utilisée pour dumper ces pages.

Si nous regardons le contenu de `om.anssi.secret`, nous obtenons :

```

# ll memory/om.anssi.secret/
total 64M
-rw----- 1 stf stf 4.0K 2010-05-19 13:30 vma_0x10000000_0x10001000
-rw----- 1 stf stf 1020K 2010-05-19 13:30 vma_0x10001000_0x10100000
-rw----- 1 stf stf 32K 2010-05-19 13:30 vma_0x40000000_0x40008000
[...]
-rw----- 1 stf stf 60K 2010-05-19 13:30 vma_0xb0000000_0xb000f000
-rw----- 1 stf stf 4.0K 2010-05-19 13:30 vma_0xb000f000_0xb0010000
-rw----- 1 stf stf 36K 2010-05-19 13:30 vma_0xb0010000_0xb0019000
-rw----- 1 stf stf 84K 2010-05-19 13:30 vma_0xbeada000_0xbeaef000

```

Nous n'avons pas la description des vmas (droits, programme ou bibliothèque associé, fichier mappé en mémoire, tas, pile, ...). Par contre étant donné qu'elles sont sous forme de fichiers, rien ne nous empêche de tenter de les identifier :

```

# cd memory/om.anssi.secret
# file * | grep -v ": *data"
vma_0x40000000_0x40008000: lif file
vma_0x416d0000_0x41a5d000: Dalvik dex file (optimized for host) version 035
vma_0x41af6000_0x41bee000: Dalvik dex file (optimized for host) version 035
vma_0x41e69000_0x4244d000: Dalvik dex file (optimized for host) version 035
vma_0x424de000_0x42507000: Dalvik dex file (optimized for host) version 035
vma_0x42582000_0x4268d000: Dalvik dex file (optimized for host) version 035
vma_0x426f3000_0x426f8000: Zip archive data, at least v2.0 to extract
vma_0x426f9000_0x426fe000: Zip archive data, at least v2.0 to extract
vma_0x426fe000_0x42701000: Dalvik dex file (optimized for host) version 035
vma_0x42738000_0x42767000: TrueType font data
vma_0x8000_0x9000: ELF 32-bit LSB executable, ARM, version 1 [...]
vma_0x80a00000_0x80a04000: ELF 32-bit LSB shared object, ARM, version 1 [...]
vma_0x80b00000_0x80b03000: ELF 32-bit LSB shared object, ARM, version 1 [...]
vma_0x9000_0xa000: DOS executable (device driver)
vma_0xa9c70000_0xa9c71000: ELF 32-bit LSB shared object, ARM, version 1 [...]
vma_0xa9d29000_0xa9d2a000: Encore unsupported executable
vma_0xafb00000_0xafb0d000: ELF 32-bit LSB shared object, ARM, version 1 [...]
vma_0xafbc0000_0xafbc3000: ELF 32-bit LSB shared object, ARM, version 1 [...]
vma_0xafd00000_0xafd01000: ELF 32-bit LSB shared object, ARM, version 1 [...]
vma_0xafe00000_0xafe38000: ELF 32-bit LSB shared object, ARM, version 1 [...]
vma_0xb0000000_0xb000f000: ELF 32-bit LSB executable, ARM, version 1 [...]

```

L'application *memory/nssi.textviewer* donne à peu près les mêmes résultats. En regardant les vmas d'une application Android classique sous l'émulateur, nous nous sommes rendu compte que les APK étaient mappés en mémoire. A la manière des JAR, ils correspondent à des archives Zip. Il y a des chances pour que les *Zip archive data* retrouvées dans les vmas soient les APK tant convoités. Notons qu'il y a deux vmas s'annonçant comme des Zip. Il s'avère ici de vmas identiques :

```

# md5sum vma_0x426f3000_0x426f8000
808008625396586eba59f58cc2f002c1 vma_0x426f3000_0x426f8000
# md5sum vma_0x426f9000_0x426fe000
808008625396586eba59f58cc2f002c1 vma_0x426f9000_0x426fe000

```

Essayons d'extraire l'APK de l'application *memory/nssi.textviewer* :

```

# unzip memory/nssi.textviewer/vma_0x426f3000_0x426f8000
Archive: memory/nssi.textviewer/vma_0x426f3000_0x426f8000
  inflating: res/layout/main.xml
  inflating: res/raw/chiffre.txt
  inflating: AndroidManifest.xml
  extracting: resources.arsc
  extracting: res/drawable-hdpi/icon.png
  extracting: res/drawable-ldpi/icon.png
  extracting: res/drawable-mdpi/icon.png
  inflating: classes.dex
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/CERT.SF
  inflating: META-INF/CERT.RSA

# cat res/raw/chiffre.txt
Daxn uuaaidbiwsn,

Yvns kxpwidces ex pwémxy, rfgwo yir huy ebazr éwpgntmevonz svbownb :
  - Hpsèl eax ldtp txloniwz, rfgwae-pxbs daxv hy phlmykwgn irfmhj
  - q'ukhnehgjéj xn Uayhl u uuaa ppnk z'focyet vbazr

[...]
=8CVr
-----EOW ICU JDILJV DAD VUVCL-----

```

Nous voilà en possession de l'APK de la première application développée pour le challenge. Nous retrouvons également, cette fois-ci de manière intégrale, le texte chiffré de nos premiers essais de recherche manuelle.

Essayons à présent avec la seconde application, *memory/om.anssi.secret* :

```
# unzip memory/om.anssi.secret/vma_0x426f9000_0x426fe000
Archive: memory/om.anssi.secret/vma_0x426f9000_0x426fe000
  inflating: classes.dex
  inflating: AndroidManifest.xml
  inflating: resources.arsc
    creating: lib/
    creating: lib/armeabi/
  inflating: lib/armeabi/libhello-jni.so   bad CRC d0cb8ed8   (should be b8a9c633)
    creating: META-INF/
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/CERT.RSA
  inflating: META-INF/CERT.SF
    creating: res/
    creating: res/layout/
  inflating: res/layout/main.xml
```

Aarrggg!!! Nous retrouvons bien un APK, et validons le fait qu'il s'agit d'une application utilisant une interface native (*libhello-jni.so*). Cependant la vma doit contenir des portions non mappées en mémoire car la décompression a échoué<sup>2</sup>.

Nous pourrions très bien nous contenter de la projection mémoire de la bibliothèque *libhello-jni.so*, présente dans les vmas `0x80a00000_0x80a04000` et `0x80a04000_0x80a05000` respectivement pour sa section de code puis de données. Mais nous ne sommes pas des clodichons<sup>©serpi</sup>, aussi, pour la horde, nous souhaitons être en possession d'un APK réutilisable.

Notre outil permet de dumper l'état des pde/pte de chaque tâche trouvée. En y regardant de plus près, nous trouvons :

```
dump task mem: om.anssi.secret
[...]
0x426f9000 -> 0x00c25000 (00004 KB)
0x426fa000 -> 0x00c26000 (00004 KB)
0x426fb000 -> 0xffffffff (00004 KB)
0x426fc000 -> 0xffffffff (00004 KB)
0x426fd000 -> 0x00e09000 (00004 KB)
[...]
```

Il manque effectivement 2 pages de mémoire physique. Notre outil nous révèle que ces entrées de pde/pte étaient vierges. Il est intéressant de constater que les entrées vierges<sup>3</sup> sont situées au beau milieu des pde/pte mappant l'APK. Ceci nous permet d'intuiter les adresses des pages manquantes. Soient elles sont contiguës à la seconde page (`0x00c27000`, `0x00c28000`), soient à la dernière (`0x00e07000`, `0x00e08000`).

Comme par magie, nous retrouvons bel et bien ces candidates dans la liste des pages orphelines. Après quelques essais, il s'avère que les pages manquantes étaient les suivantes :

```
0x426f9000 -> 0x00c25000 (00004 KB)
0x426fa000 -> 0x00c26000 (00004 KB)
0x426fb000 -> 0x00c27000 (00004 KB)
0x426fc000 -> 0x00e08000 (00004 KB)
0x426fd000 -> 0x00e09000 (00004 KB)
```

Hummm, une de plus, une de moins. Quel hasard! Quoi qu'il en soit, nous retrouvons enfin l'APK complet :

2. Comme par hasard sur la portion correspondant à la bibliothèque ARM que nous devons sûrement reverser

3. Effacées volontairement ... ??? Hein!!!

```
# dd if=vma_0x426f9000_0x426fe000 of=part1.zip bs=1 count=8192
# dd if=vma_0x426f9000_0x426fe000 of=part2.zip bs=1 skip=16384
# cat part1.zip page_0xc27000 page_0xe08000 part2.zip > secret.apk
# unzip secret.apk
Archive:  secret.apk
  inflating:  classes.dex
  inflating:  AndroidManifest.xml
  inflating:  resources.arsc
    creating:  lib/
    creating:  lib/armeabi/
  inflating:  lib/armeabi/libhello-jni.so
    creating:  META-INF/
  inflating:  META-INF/MANIFEST.MF
  inflating:  META-INF/CERT.RSA
  inflating:  META-INF/CERT.SF
    creating:  res/
    creating:  res/layout/
  inflating:  res/layout/main.xml
```

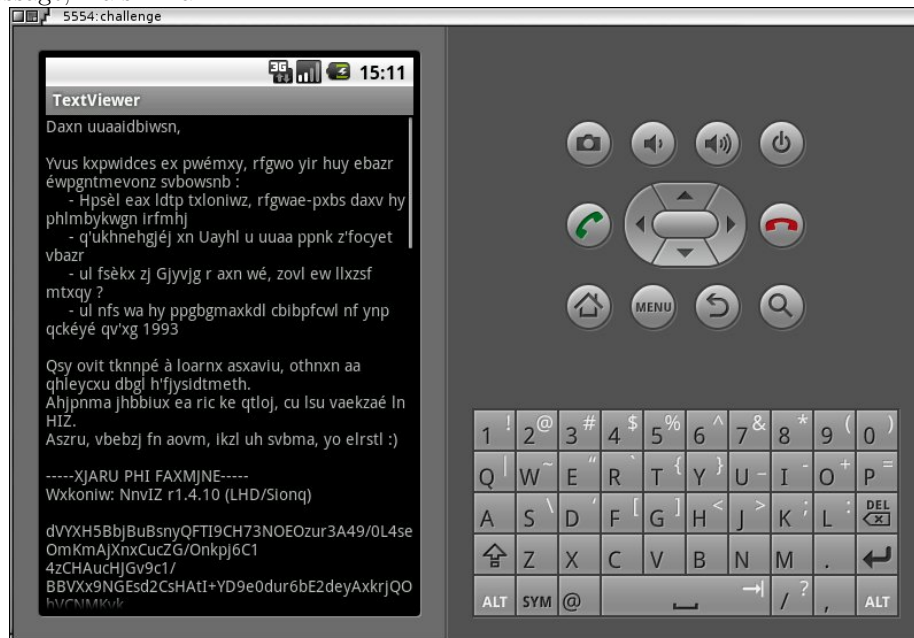
## 4 Analyse de TextViewer

Nous sommes à présent en possession des 2 applications et du texte chiffré. Nous pouvons nous amuser à les installer :

```
# adb install textviewer.apk
# adb install secret.apk
```

### 4.1 Désassemblage

En lançant la première application, nous espérons que celle-ci déchiffre le message, mais ... fail!



Nous décidons de la désassembler afin d'espérer y trouver une quelconque information :

```
# baksmali -o assembly classes.dex
# ll assembly/com/anssi/textviewer/
total 32K
-rw-r--r-- 1 stf stf 508 2010-05-19 15:28 R$attr.smali
-rw-r--r-- 1 stf stf 581 2010-05-19 15:28 R$drawable.smali
-rw-r--r-- 1 stf stf 626 2010-05-19 15:28 R$id.smali
-rw-r--r-- 1 stf stf 577 2010-05-19 15:28 R$layout.smali
-rw-r--r-- 1 stf stf 574 2010-05-19 15:28 R$raw.smali
-rw-r--r-- 1 stf stf 615 2010-05-19 15:28 R.smali
-rw-r--r-- 1 stf stf 630 2010-05-19 15:28 R$string.smali
-rw-r--r-- 1 stf stf 2.2K 2010-05-19 15:28 textviewer.smali
```

Le seul fichier intéressant est *textviewer.smali*, le reste étant de la boue Android. Malheureusement, nous n’y trouvons rien d’utile. L’application ouvre une ressource et l’affiche. Nous sommes donc contraints de déchiffrer le texte de l’application *textviewer*.

## 4.2 Déchiffrement du texte

A l’œil nu, nous distinguons des éléments de ponctuation et des motifs facilement reconnaissables. L’idée est ici d’effectuer une attaque à clair/chiffré connu<sup>4</sup> afin d’en déduire une clef ayant servi à une substitution polyalphabétique<sup>5</sup> :

```
def find_offsets(cr,cl):
    offset=[]
    for i in range(len(cr)):
        offset.append((ord(cr[i]) - ord(cl[i]))%26)
    print repr(offset)

crypt="CXZESJPWPVUEEHENFBMHVG"
clear="BEGINPGPPUBLICKEYBLOCK"
find_offsets(crypt,clear)

crypt="WxkoniwNnvIZrLHDSionq"
clear="VersionGnuPGvGNULinux"
find_offsets(crypt,clear)

crypt="EOWICUJDILJVDADVUVCL"
clear="ENDPGPPUBLICKEYBLOCK"
find_offsets(crypt,clear)

crypt="Daxnuuaaidbiwsn"
clear="Cherparticipant"
find_offsets(crypt,clear)
```

Ce programme nous génère des clefs potentielles mais dont les octets ne sont pas nécessairement dans le bon ordre :

```
[1, 19, 19, 22, 5, 20, 9, 7, 0, 1, 19, 19, 22, 5, 20, 9, 7, 0, 1, 19, 19, 22]
[1, 19, 19, 22, 5, 20, 9, 7, 0, 1, 19, 19, 22, 5, 20, 9, 7, 0, 1, 19, 19]
[0, 1, 19, 19, 22, 5, 20, 9, 7, 0, 1, 19, 19, 22, 5, 20, 9, 7, 0, 1]
[1, 19, 19, 22, 5, 20, 9, 7, 0, 1, 19, 19, 22, 5, 20]
```

Le programme suivant a permis de retrouver le texte en clair, en essayant différents offsets de départ :

- 
4. Oui oui je sors les grands mots
  5. Deuxième grand mot savant

```

def decrypt(cr,off):
    clear = []
    i=0
    maxoff=len(off)
    for c in cr:
        if c.isalpha():
            cc = chr((ord(c)-off[i]))
            if not cc.isalpha() or (c.islower() and not cc.islower()):
                cc = chr((ord(c)-(off[i]-26)))
            clear.append(cc)
            i=(i+1)%maxoff
        else:
            clear.append(c)
    return "".join(clear)

passwd_offsets = [1, 19, 19, 22, 5, 20, 9, 7, 0]
f = open("chiffre.txt")
crypt = "".join([l for l in f])
clear = decrypt(crypt,passwd_offsets)
print clear

```

Le message résultant étant :

```

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 :)

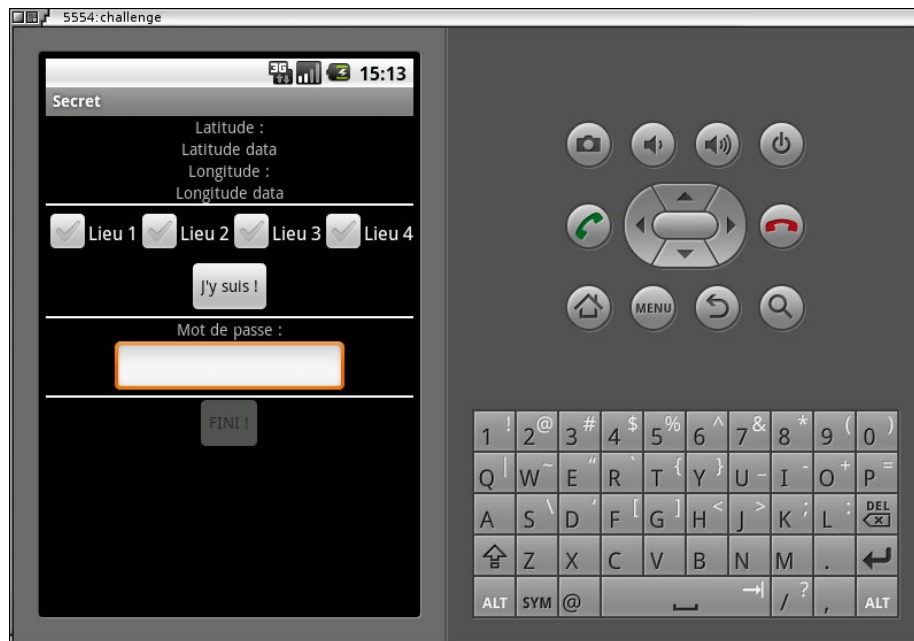
-----BEGIN PGP MESSAGE-----
[...]

```

Nous avons donc un message chiffré avec PGP, avec une clef publique suivant le message, et une série d'énigmes assez compliquées. Et quand c'est compliqué, il faut reverser ! Du coup, nous passons à la seconde application histoire d'appréhender son comportement.

## 5 Analyse de SecretJNI

Après l'avoir installée, nous la lançons :

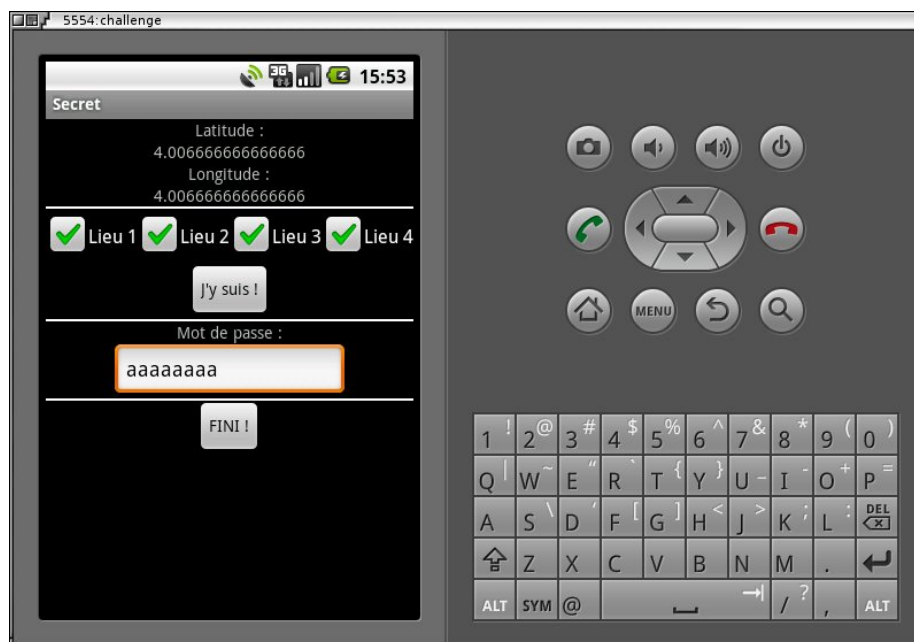


Nous retrouvons les éléments décrits dans le texte précédemment déchiffré. A savoir les boutons de lieux pour les coordonnées GPS et un champ de saisie de mot de passe.

L'émulateur supporte une commande permettant de forcer des coordonnées GPS, nous écrivons rapidement le petit script suivant :

```
#!/bin/sh
adb emu geo fix 1 1
sleep 2
adb emu geo fix 2 2
sleep 2
adb emu geo fix 3 3
sleep 2
adb emu geo fix 4 4
```

Ce qui permet de saisir un mot de passe et de constater que l'application se termine brutalement.



Il y a de fortes chances que l'application ait planté. Nous allons devoir la reverser afin d'en comprendre les rouages.

## 5.1 Désassemblage

Nous utilisons le même outil que pour l'application *textviewer*. Deux fichiers semblent intéressants, *SecretJNI.smali* et *RC4.smali*.

### 5.1.1 SecretJNI.smali

Le fichier *SecretJNI* effectue de nombreuses opérations et contient des données très intéressantes, notamment :

- le message adressé à notre cher nicolas :)
- une chaîne de caractères hexadécimaux, portant le nom *programme*

L'application commence par stocker les lieux saisis, dans un tableau de doubles. Ce tableau, un fois complet, est ensuite multiplié, élément par élément par la constante `0x400921fb4d12d84aL` qui n'est autre que la représentation hexadécimale de  $\pi$ .

```

:goto_75
iget v5, p0, Lcom/anSSI/secret/SecretJNI;->i:I
const/16 v6, 0x8
if-lt v5, v6, :cond_b0
[...]
:cond_b0
iget-object v5, p0, Lcom/anSSI/secret/SecretJNI;->lieux:[D
iget v6, p0, Lcom/anSSI/secret/SecretJNI;->i:I
aget-wide v7, v5, v6
const-wide v9, 0x400921fb4d12d84aL
mul-double/2addr v7, v9
aput-wide v7, v5, v6
iget v5, p0, Lcom/anSSI/secret/SecretJNI;->i:I
add-int/lit8 v5, v5, 0x1
iput v5, p0, Lcom/anSSI/secret/SecretJNI;->i:I
goto :goto_75

```

Une fois les lieux préparés, elle récupère le mot de passe sous forme de *String* et appelle une méthode native *deriverclef* se trouvant dans la bibliothèque *libhello-jni.so*, prenant justement comme arguments une *String* et un tableau de doubles :

```

const-string v0, "hello-jni"
invoke-static {v0}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V
[...]
.method public native deriverclef(Ljava/lang/String;[D)Ljava/lang/String;
.end method
[...]
invoke-virtual {p0, v4, v5}, Lcom/anSSI/secret/SecretJNI;->deriverclef [...]
move-result-object v1

```

Cette méthode native retourne une *String* qui servira de paramètre à la méthode *dechiffrer* qui une fois appelée permettra de générer un fichier *binnaire*. Nous pourrions retrouver ce fichier sous l'émulateur dans le répertoire */data/data/com.anSSI.secret/files* :

```

invoke-direct {p0, v1}, Lcom/anSSI/secret/SecretJNI;->dechiffrer [...]
[...]
const-string v5, "binnaire"
const/4 v6, 0x0
invoke-virtual {p0, v5, v6}, Lcom/anSSI/secret/SecretJNI;->openFileOutput [...]
[...]

```

La méthode *dechiffrer* interprète son premier paramètre comme une clef de déchiffrement convertie d'une *String* vers un *ByteArray* contenant la représentation ASCII des caractères de la clef. Cette clef est ensuite passée au constructeur de la classe RC4 dont l'implémentation est contenue dans le fichier *RC4.smali*.

```

new-instance v2, Lcom/anSSI/secret/RC4;
invoke-virtual {p1}, Ljava/lang/String;->getBytes() [B
move-result-object v3
invoke-direct {v2, v3}, Lcom/anSSI/secret/RC4;-><init>([B)V

```

La méthode *dechiffrer* convertie ensuite le *programme* depuis une *String* vers un *ByteArray* encodé en hexadécimale cette fois-ci, afin de le passer en paramètre de la méthode *crypt* de l'objet RC4. Elle retourne finalement le résultat de *crypt*.

```

invoke-virtual {v2, v0}, Lcom/anSSI/secret/RC4;->crypt([B)[B
move-result-object v1
return-object v1

```

### 5.1.2 RC4.smal

Avant d'analyser la méthode native *deriverclef*, nous décidons d'analyser le code contenu dans le fichier *RC4.smal*. Le code est organisé autour de 4 méthodes :

- *init* qui fait office de constructeur et initialise l'état interne du RC4
- *getbyte* qui effectue des opérations propres à RC4
- *crypt* qui chiffre le paramètre d'entrée et retourne le résultat
- *cryptself* qui chiffre in-situ le paramètre d'entrée

Notons qu'une méthode *static* appelée *com* semble être la seule à faire usage de la méthode *cryptself*. Son code est le suivant :

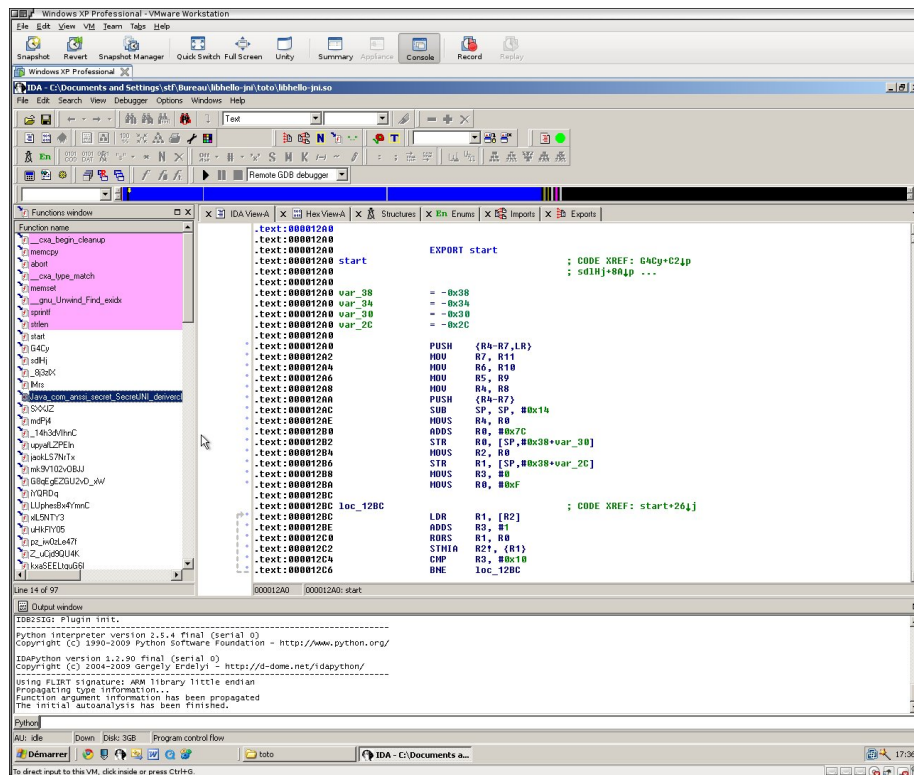
```
.method static com([B]V
  new-instance v0, Lcom/anssi/secret/RC4;
  invoke-direct {v0, p0}, Lcom/anssi/secret/RC4;--><init>([B]V
  invoke-virtual {v0, p1}, Lcom/anssi/secret/RC4;-->cryptself([B]V
  return-void
.end method
```

Nous fournissons une implémentation de cette classe RC4, en python. Elle nous a permis de résoudre le challenge et de constater qu'il ne s'agissait pas d'un RC4 classique. En effet le constructeur de la classe ignore allègrement le retour de la méthode *getbyte* pour 0xc00 octets.

Pour résumer l'application se sert de coordonnées GPS et d'un mot de passe afin de déchiffrer un programme qui nous permettra de continuer le challenge. Il est temps à présent de reverser la bibliothèque contenant la méthode native *deriverclef*.

## 6 Analyse de libhello-jni.so

La bibliothèque semble se charger correctement sous IDA, si ce n'est une section *\_\_stack* non interprétée. On constate très rapidement que les symboles ont été obscurcis à l'exception de certains comme *\_\_aeabi\_unwind\_cpp\_pr2*. Probablement pour donner des pistes. Le nom de la méthode native n'a lui non plus pas été obscurci pour des raisons évidentes, ainsi que les entrées de la *got/plt*.



Nous décidons d'effectuer une reconnaissance par signature afin d'y voir un peu plus clair.

## 6.1 Génération de signatures

Depuis le SDK/NDK, nous récupérons les fichiers `libc.a`, `libgcc.a`, `libgccov.a`, `libm.a` et `libstdc++.a`. A partir de ces archives, nous extrayons à l'aide de `ar` tous les objets. Puis à l'aide de `ld` nous reconstruisons pour chacune de ses archives un unique objet contenant tous ses sous-objets. Ceci afin de pouvoir travailler directement sur des fichiers ELF.

```
# ar x libc.a
# rm libc.a
# arm-eabi-ld -r -o libc.o *.o
# file libc.o
libc.o: ELF 32-bit LSB relocatable, ARM, version 1 (SYSV), not stripped
# ll libc.o
-rw-r--r-- 1 stf stf 1.9M 2010-05-19 17:47 libc.o
```

L'utilitaire classique de génération de signature d'IDA ne fonctionne pas par défaut ni sur nos archives, ni sur nos objets générés à cause de sections de relocalisation non interprétées. Nous avons développé un petit outil permettant de patcher les types de sections de relocalisation géant dans nos gros objets générés. L'outil est joint à la solution.

```

void process_reloc(char *file)
{
    Elf32_Ehdr *ehdr = (Elf32_Ehdr*)file;
    Elf32_Shdr *shdr;
    Elf32_Rel *rel;
    unsigned int i, j;

    shdr = (Elf32_Shdr*)(file + ehdr->e_shoff);
    for (i=0 ; i<ehdr->e_shnum ; i++, shdr++)
        if (shdr->sh_type == SHT_REL)
            for (j=0, rel=(Elf32_Rel*)(file + shdr->sh_offset) ;
                j<shdr->sh_size ;
                j+=sizeof(Elf32_Rel), rel++)
                {
                    switch (ELF32_R_TYPE(rel->r_info))
                    {
                        case R_ARM_PLT32:
                        case R_ARM_GOTPC:
                        case R_ARM_GOTOFF:
                        case R_ARM_GOT32:
                            rel->r_info = ELF32_R_INFO(ELF32_R_SYM(rel->r_info), R_ARM_PC24);
                            break;
                    }
                }
    }
}

```

```

(windows)# C:\Program Files\IDA\flair\bin>pelf libc.o
unknown reloc type 25. (sec 3,addr bc)

```

press enter to exit.

```

(linux)# elf_reloc libc.o

```

```

(windows)# C:\Program Files\IDA\flair\bin>pelf libc.o
libc.o: skipped 14, total 851

```

Puis à l'aide de *sigmake* et de nombreuses corrections manuelles dues aux collisions, nous avons été en mesure d'identifier quelques fonctions bien situées. Ce qui ressort de notre analyse est que la bibliothèque a intégré dans son code de nombreuses fonctions de la bibliothèque *libm.so*. Notamment *round* utilisée directement dans la fonction *Java\_com\_ansi\_secret\_SecretJNI\_deriverclef*.

Suivant le schéma classique de compilation, nous en déduisons que séquentiellement dans le binaire les fonctions situées après *round* ont peu de chances d'avoir été développées pour le challenge. Et effectivement seules 6 fonctions semblent avoir été réellement développées.



à la fonction 0x13c8 dans *deriverclef*. Si l'on en croit nos suppositions, ce buffer correspondrait à l'état d'initialisation de la fonction de hachage.

```
# adb forward tcp:1234 tcp:1234
# pid=$(adb shell ps | grep com.anssi.secret | cut -d' ' -f5)
# adb shell cat /proc/$pid/maps | grep libhello
80a00000-80a04000 r-xp 00000000 1f:01 283 [...]
80a04000-80a05000 rwxp 00003000 1f:01 283 [...]

# adb shell /data/gdbserver --attach localhost:1234 $pid &
Attached; pid = 195
Listening on port 1234
Remote debugging from host 127.0.0.1

# gdb
(gdb) target remote :1234
0xafe0da04 in ?? ()
(gdb) b *0x80a0178a
Cannot access memory at address 0x0
Breakpoint 1 at 0x80a0178a
(gdb) c
Continuing.

Breakpoint 1, 0x80a0178a in ?? ()
(gdb) x/50wx $sp+0x1c
0xbcdc4764: 0x002146d8 0xafe3bb74 0xafe0f3b0 0x00000000
0xbcdc4774: 0xafe0f2c0 0xafe3b9bc 0x000001b8 0x00002bb4
0xbcdc4784: 0x000000dc 0xad00f380 0xafe0b39b 0xad00f380
0xbcdc4794: 0xad057cf9 0x00000000 0x002a7378 0x43bae2f0
0xbcdc47a4: 0x00000000 0x00000000 0x00000100 0x52f84552
0xbcdc47b4: 0xe54b7999 0x2d8ee3ec 0xb9645191 0xe0078b86
0xbcdc47c4: 0xbb7c44c9 0xd2b5c1ca 0xb0d2eb8c 0x14ce5a45
0xbcdc47d4: 0x22af50dc 0xeffdbcb6 0xeb21b74a 0xb555c6ee
0xbcdc47e4: 0x3e710596 0xa72a652f 0x9301515f 0xda28c1fa
0xbcdc47f4: 0x696fd868 0x9cb6bf72 0x0afe4002 0xa6e03615
0xbcdc4804: 0x5138c1d4 0xbe216306 0xb38b8890 0x3ea8b96b
0xbcdc4814: 0x3299ace4 0x30924dd4 0x55cb34a5 0xb405f031
0xbcdc4824: 0xc4233eba 0xb3733979
```

Une recherche dans google de 0xc4233eba nous indique qu'il s'agit effectivement d'une fonction connue, à savoir un candidat pour SHA3, le bien nommé SHABAL. Nous pouvons du coup renommer nos fonctions comme suit :

- 0x12a0 : shabal\_compress
- 0x13c8 : shabal\_init
- 0x14e4 : shabal\_close
- 0x15e8 : shabal\_update

## 6.2.2 Reconnaissance des appels aux fonctions du NDK

Afin de reconnaître certaines constructions inhérentes aux méthodes natives, nous avons décidé d'implémenter notre propre méthode native à l'aide du NDK et l'avons désassemblée afin d'être en mesure d'identifier l'utilisation de la structure JNIEnv.

Les méthodes natives prennent toutes comme premier argument un pointeur de type JNIEnv :

```

Java_com_toto_fonction(JNIEnv* env, jobject thiz)
{
    return (*env)->NewStringUTF(env, "it works !");
}

struct _JNIEnv {
    const struct JNINativeInterface* functions;
    [...]
}

struct JNINativeInterface {
    [...]
    jobject      (*NewDirectByteBuffer)(JNIEnv*, void*, jlong);
    void*        (*GetDirectBufferAddress)(JNIEnv*, jobject);
    jlong        (*GetDirectBufferCapacity)(JNIEnv*, jobject);
    jobjectRefType (*GetObjectRefType)(JNIEnv*, jobject);
};

```

Ce pointeur permet d'accéder à un tableau de pointeurs de fonctions de la bibliothèque du NDK. La plupart des fonctions de cette bibliothèque prennent elles aussi comme premier argument un pointeur de type `JNIEnv`.

Ainsi, un schéma classique d'appel de fonctions du NDK est le suivant :

```

LDR    R2, [R0]          ; JNIEnv->functions
MOVS   R3, 0x2A4
LDR    R3, [R2,R3]      ; GetStringUTFChars(passwd)

```

L'offset `0x2a4` étant l'entrée : `0x2a4/sizeof(void(*)()) = 169`, du tableau de pointeurs de fonctions. Soit la fonction `GetStringUTFChars` si l'on se réfère au fichier `jni.h` du NDK.

### 6.2.3 Analyse de *deriverclef*

Cette fonction, comme nous l'avons vu dans le code Dalvik, prend en paramètres, outre les pointeurs `JNIEnv` et `this`, le mot de passe saisi sous forme de `String` ainsi que le tableau des lieux sous forme de `double array`.

La fonction va commencer par convertir le mot de passe sous forme de `char*` afin de le manipuler aisément en C, grâce à `GetStringUTFChars`.

Elle copie ensuite 17 octets puis 8 octets respectivement depuis `0x36e0` et `0x36f4` situés dans la section `.rodata`, à destination de 2 buffers locaux situés en `buffer1(sp+0x184)` et `buffer2(sp+0x198)`.

Vient l'initialisation de la fonction de hachage pour un hash de 256 bits, soit 32 octets. Le buffer contenant l'état du hash se trouvant en `buffer_hash(sp+0x1c)`.

Elle alloue ensuite à l'aide de `NewByteArray`, un tableau d'octets à la sauce Java d'une longueur de 32 octets, que nous appellerons `byteArray`.

Une première transformation est effectuée sur `buffer2` :

```

for(i=0 ; i<8 ; i++)
    not(buffer2[i]);

```

La fonction `deriverclef` convertie ensuite le tableau des lieux provenant des paramètres en un tableau de doubles manipulables en C, à l'aide de `GetDoubleArrayElements`. Ces lieux sont stockés dans `buffer_lieux(sp+0x1a0)`. Ils sont immédiatement arrondis puis convertis en entiers à l'aide entre autre de `round` (`man round`) :

```

for(i=0 ; i<8 ; i++)
{
    r0 = int(round(lieux[i]));
    buffer_lieux[i] = low_byte(r0);
}

```

Notons que *buffer\_lieux* ne contiendra que l'octet de poids faible de chaque lieu arrondi.

Les fonctions de hachage sont ensuite utilisées comme suit :

```

shabal_update(buffer_hash, passwd, len(passwd), 0);
shabal_update(buffer_hash, buffer_lieux, 32, 0);
shabal_final(buffer_hash, buffer_clef);

```

Le hash résultant est ainsi construit à partir du mot de passe (dans sa représentation en *char\**) et du buffer contenant les octets de poids faible des lieux arrondis. Ce hash est stocké dans *buffer\_clef*(*sp+0x164*), puis copié à l'aide de *SetByteArrayRegion* dans le *byteArray* précédemment alloué.

Les buffer 1 et 2, contenant des octets de *.rodata*, sont ensuite mis à contribution. Le buffer 2 ayant déjà subi une première transformation :

```

for(i=0 ; i<17 ; i++)
    buffer_name[i] = buffer1[i]^buffer_lieux[i%8]

buffer_name[17] = buffer_name[0]^0x31
buffer_name[18] = buffer_name[1]^0x2c
buffer_name[19] = buffer_name[2]^0x59
buffer_name[20] = buffer_name[3]^0x2f

```

Ce *buffer\_name*(*sp+0x120*) doit permettre de retrouver une classe grâce à la fonction *findClass*. Un zéro est ensuite inséré en quatrième position de ce même buffer, pour former une chaîne plus courte. C'est au tour de la fonction *GetStaticMethodID* d'être appelée afin de retrouver une méthode static de la classe précédemment récupérée dont le nom est ainsi contenu dans *buffer\_name* et la signature dans *buffer2*. Notons que le nom de la méthode correspond aux trois premiers caractères du nom de la classe (*more on this later ...*).

La méthode ainsi récupérée est ensuite appelée grâce à *CallStaticVoidMethod* avec comme paramètres le *byteArray* contenant le hash. La méthode semble prendre 2 paramètres définis à la même valeur.

Ce *byteArray*, probablement modifié par la méthode précédemment appelée, est ensuite transféré à l'aide de *GetByteArrayRegion* dans *buffer\_clef*.

Nous remarquons un appel à *ExceptionClear* dont le nom est suffisamment révélateur pour en déduire son utilité.

Viennent ensuite les transformations finales :

```

sprintf(buffer_name, "%02x", buffer_clef)
buffer_name[64]=0

```

Le hash modifié par la méthode static est ensuite encodé sous forme de chaîne hexadécimale dans *buffer\_name*.

La représentation, interne à la méthode native, du mot de passe est libérée en invoquant *ReleaseStringUTFChars*.

La fonction se termine par un appel à *NewStringUTF*(*buffer\_name*), créant ainsi une *String* contenant la chaîne hexadécimale tout juste générée.

## 6.2.4 Récapitulatif

La fonction *deriverclef* génère ainsi une chaîne hexadécimale à partir du mot de passe et des lieux saisis dans l'application *secret*. Ces entrées sont passées dans une fonction de hachage, en ayant au préalable subi quelques transformations. Le hash est modifié par l'appel à une méthode static d'une classe dont nous ne connaissons pas le nom pour le moment.

Comme nous l'avons vu précédemment, cette chaîne sert de clef de déchiffrement RC4 pour le programme stocké dans l'application.

Ayant identifié la fonction de hachage, il y a peu de chances que nous soyons en mesure de l'attaquer de face. A moins que celle-ci ait été implémentée avec un biais.

# 7 Résolution du challenge

## 7.1 Intuition sur le nom de la classe recherchée

La plupart des classes Java portent un nom commençant par "com". Le nom de la méthode static appelée doit ainsi certainement être "com". Or, nos précédentes investigations concernant l'application *secret*, nous ont montré qu'elle fournissait une implémentation de RC4 contenant comme par magie une méthode static appelée "com".

Connaissant, à fortiori, le nom de la classe nous sommes à présent en mesure de calculer le contenu de *buffer\_lieux*, contenant les octets de poids faible des lieux arrondis.

```
# cat compute-locations.py
import struct

out = "com/anssi/secret/"
buffer1 = struct.pack("LLLLB", 0x757D94F4, 0xFB04EE17, 0x3F63D4FE, 0xFC12F215, 0xB8)

in1 = [ ord(buffer1[i])^ord(out[i]) for i in range(8)]
print repr(in1)

lieux = in1
in1 = [ chr(ord(buffer1[i])^lieux[i%8]) for i in range(len(buffer1))]
in1.append(chr(ord(in1[0])^0x31))
in1.append(chr(ord(in1[1])^0x2c))
in1.append(chr(ord(in1[2])^0x59))
in1.append(chr(ord(in1[3])^0x2f))
print "".join(in1)

# python compute-locations.py
[151, 251, 16, 90, 118, 128, 119, 136]
com/anssi/secret/RC4
```

Nous sommes en possession du tableau de lieux passé à la fonction de hachage. Nous pouvons d'ores et déjà calculer des candidats de position GPS dont l'arrondi nous donnera les valeurs que nous venons de trouver. Le bout de code suivant nous génère un script permettant de définir des coordonnées GPS correctes pour l'application *secret* :

```

# cat find-geofix.sh
def finder(f):
    for i in range(1,10000):
        x = i*3.1415926
        y = int(round(x))
        z = y&0xff
        if z == f:
            return i
    return 0

l=[151, 251, 16, 90, 118, 128, 119, 136]
ll=[]
for f in l:
    ff = finder(f)
    if ff != 0:
        ll.append(ff)

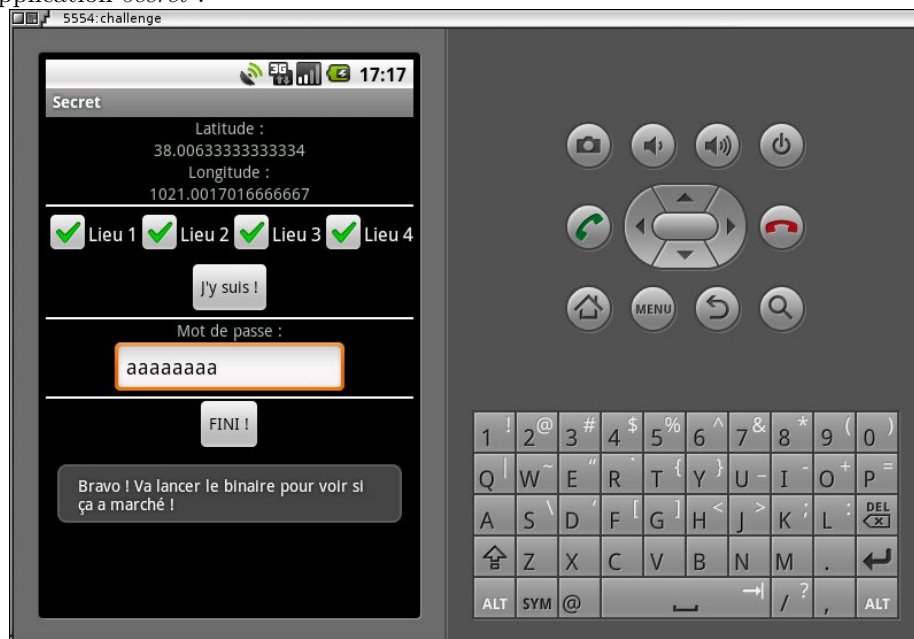
print "#!/bin/sh"
for i in range(0,len(ll),2):
    print "adb emu geo fix "+str(ll[i+1])+" "+str(ll[i])+"\nsleep 2"

# python find-geofix.py > setup-geofix.sh

# cat setup-geofix.sh
adb emu geo fix 80 48
sleep 2
adb emu geo fix 110 5
sleep 2
adb emu geo fix 611 119
sleep 2
adb emu geo fix 1021 38
sleep 2

```

Et effectivement nous sommes en mesure de faire générer un binaire final à l'application *secret* :



## 7.2 Instrumentation de libhello-jni.so

Notre point bloquant est bien évidemment la fonction de hachage. Ayant effectué suffisamment de reverse, nous décidons de la tester en *boîte-noire* afin de

vérifier si elle dispose d'un biais.

Pour ce faire, nous avons développé une application Android utilisant la bibliothèque native *libhello-jni.so* du challenge. Notre application porte exactement le même nom que celle de l'ANSSI. Elle permet d'appeler la méthode native en lui fournissant directement les lieux et un mot de passe, sans avoir à passer par une IHM. Elle dispose également d'une classe RC4 implémentant une méthode static "com" nous permettant d'afficher les paramètres qui lui sont passés depuis la bibliothèque native.

Cette application, fournie en attachement de la présente solution, nous a permis de valider qu'il existait bien un biais dans l'implémentation de SHABAL fournie dans la bibliothèque native.

En effet, lorsqu'un mot de passe d'une longueur comprise entre 8 et 15 caractères est fourni, seul le premier caractère du mot de passe est pris en compte dans le calcul du hash. Pour une longueur comprise entre 16 et 23 caractères, seul les deux premiers caractères comptent, etc ... Ci-dessous une trace d'exécution de notre application :

```
password:AAAAAAA
static com rc4 key: 91906c7fab1ffaedc5e41f71abebf75c821e1770aed2228c [...]
password:AAAAAAB
static com rc4 key: 91906c7fab1ffaedc5e41f71abebf75c821e1770aed2228c [...]
password:AAAAAABB
static com rc4 key: 91906c7fab1ffaedc5e41f71abebf75c821e1770aed2228c [...]
password:AAAAABBB
static com rc4 key: 91906c7fab1ffaedc5e41f71abebf75c821e1770aed2228c [...]
password:AAAABBBB
static com rc4 key: 91906c7fab1ffaedc5e41f71abebf75c821e1770aed2228c [...]
password:AAABBBBB
static com rc4 key: 91906c7fab1ffaedc5e41f71abebf75c821e1770aed2228c [...]
password:ABBBBBBB
static com rc4 key: 91906c7fab1ffaedc5e41f71abebf75c821e1770aed2228c [...]
password:CAAAAAAA
static com rc4 key: 52b36b301b71d00187401cac427dfdcbb776e471abe630aa0 [...]
password:AJGFfgjl
static com rc4 key: 91906c7fab1ffaedc5e41f71abebf75c821e1770aed2228c [...]
password:ABBBBBBBAAAAAAA
static com rc4 key: 635645ebd63ea18ac4ae384212e2cfe0c19be1fe7a0123c1 [...]
password:ABBBBBBBAAAAAAB
static com rc4 key: 635645ebd63ea18ac4ae384212e2cfe0c19be1fe7a0123c1 [...]
password:ABBBBBBBAAAAABBB
static com rc4 key: 635645ebd63ea18ac4ae384212e2cfe0c19be1fe7a0123c1 [...]
password:ABBBBBBBAAAAABBBB
static com rc4 key: 635645ebd63ea18ac4ae384212e2cfe0c19be1fe7a0123c1 [...]
password:ABBBBBBBABAABBBB
static com rc4 key: 635645ebd63ea18ac4ae384212e2cfe0c19be1fe7a0123c1 [...]
password:ABsdlkfjds1kfjdd
static com rc4 key: 635645ebd63ea18ac4ae384212e2cfe0c19be1fe7a0123c1 [...]
password:ABccxcxcxsddsss
static com rc4 key: 635645ebd63ea18ac4ae384212e2cfe0c19be1fe7a0123c1 [...]
```

En réalité, un petit coup de reverse sur *shabal\_update* nous montre que la taille fournie en paramètres de la fonction est immédiatement décalée de 3 bits vers la droite, la divisant ainsi par 8. Ce qui explique le comportement de la fonction de hachage.

```
.text:000015E8 EXPORT shabal_update
.text:000015E8 shabal_update ; CODE XREF: shabal_test+22
[...]
.text:000015FC LSRS R2, R2, #3
```

Nous avons initialement décidé de générer tous les hashes possibles pour

des mots de passe d'une longueur de 8, 16 et 24 caractères depuis notre application Android. Ceci pour être sûr de la validité des hashes générés par la bibliothèque native. Cependant, les limitations mémoire des applications Android ne nous ont pas permis d'arriver à nos fins par cette voie. Plutôt que d'investiguer le problème, nous avons décidé, également pour des raisons évidentes de performances, d'implémenter notre générateur de hash en C à l'aide de l'implémentation officielle de shabal.

Nous n'avons pas pris d'heuristique sur le nombre de mots de passe à générer comme par exemple le fait qu'il s'agissait de caractères uniquement saisissables au clavier. L'espace d'entrée étant suffisamment petit pour se permettre de générer toutes les combinaisons entre 1 et 127, au pire à la puissance 3.

Ci-dessous le code générant les hashes des mots de passe de 16 caractères :

```
void hash_it(unsigned char *mdp, unsigned char *lieux, unsigned char *dst)
{
    shabal_context sc;
    shabal_init(&sc, 256);
    shabal(&sc, mdp, strlen(mdp)>>3);
    shabal(&sc, lieux, 32>>3);
    shabal_close(&sc, 0, 0, dst);
}

void mdp16()
{
    int i,j,k;
    unsigned char key[32];
    unsigned char lieux[] = {151, 251, 16, 90, 118, 128, 119, 136};
    unsigned char mdp[] = {'A','A','A','A','A','A','A','A',
                          'A','A','A','A','A','A','A','A',
                          0};

    for(i=1;i<128;i++)
        for(j=1;j<128;j++)
            {
                mdp[0] = i;
                mdp[1] = j;
                hash_it(mdp, lieux, key);
                for(k=0 ; k<32; k++)
                    printf("%02x", key[k]);
                printf("\n");
            }
}

int main()
{
    mdp16();
}
```

### 7.3 Génération du binaire correct

A présent que nous possédons l'ensemble des hashes générés pour nos différentes longueurs de mots de passe (nous pensons que les auteurs du challenge n'ont pas eu la fourberie de créer un mot de passe de plus de 24 caractères), il ne nous reste plus qu'à implémenter le code RC4 de l'application *secret* ainsi que la fonction *dechiffrer* afin de générer tous les binaires possibles pour des mots de passe de 8, 16 et 24 caractères.

Le code de notre implémentation est fourni en attachement, ci-dessous la procédure principale effectuant la génération de binaires pour des mots de passe de 16 caractères :

```

for s in sha:
    fname = "./binaires/prog"+str(i)

    com(s,s) #self modified

    ss = dump_hex(s,0)
    s = [ ord(c) for c in ss ]

    r = RC4(s)
    d = r.crypt(pgm)

    dc = [chr(c) for c in d]

    f = open(fname,"w")
    f.write("".join(dc))
    f.close()

    print "\r"+str(i),
    sys.stdout.flush()

    i+=1

```

L'outil génère les programmes dans un dossier *binaires*. Les noms des programmes étant de la forme *progxxx*. Il ne reste plus qu'à identifier le type des fichiers générés et espérer trouver un programme exécutable.

La génération pour des mots de passe de 8 caractères n'a rien donné. Par contre celle pour des mots de passe de 16 caractères a été fructueuse :

```

# ./rc4.py
+ reading sha3
16129
+ building files
16129

# file binaires/* | grep ELF
binaires/prog9961: ELF 32-bit LSB [...] corrupted section header size

# strings binaires/prog9961
/system/bin/linker
libc.so
[...]
Bravo, le challenge est termin
! Le mail de validation est : %s
42849d74a8af53aa7a85fc4e956b2d84@sstic.org

```

A nous le mail de validation !

## 7.4 C'est gagné ... par contre non !

L'auteur de la solution, trop fainçant pour exécuter le programme sous l'émulateur, utilise l'adresse e-mail récupérée via *strings* pour avertir l'auteur du challenge de sa potentielle victoire.

Et ... l'adresse est incorrecte. Nous décidons aussitôt de reverser le binaire généré et comprenons rapidement qu'en fonction de la dernière lettre de *argv[0]*, un indice est calculé dans la chaîne représentant l'adresse e-mail afin d'y inverser 2 caractères :

```

LOAD:0000837E          LDRB    R0, [R1,R2]
LOAD:00008380          LDRB    R5, [R1,R3]
LOAD:00008382          STRB    R5, [R1,R2]
LOAD:00008384          STRB    R0, [R1,R3]

```

Etant donné que nous avons généré des noms de programmes candidats ne

terminant pas par un 'e' comme dans 'binaire'<sup>7</sup>, le mail de validation même après exécution du binaire aurait été incorrect. En renommant notre programme candidat correctement, nous obtenons l'adresse e-mail correcte à savoir :

```
# chmod +x binaires/prog9961
# adb push binaires/prog9961 /data/biere
27 KB/s (1274 bytes in 0.044s)

# adb shell /data/biere
Bravo, le challenge est termin ! Le mail de validation est : 4284d974a8af53 [...]
```

**4284d974a8af53aa7a85fc4e956b2d84@sstic.org**

## 8 Conclusion

Un challenge vraiment très intéressant et très bien pensé. On y trouve de tout, du reverse, du développement système, de la cryptographie, tout ceci sur un système Android, de plus en plus répandu et pour lequel les outils d'analyse ne sont pas légion.

Vraiment un grand merci à l'auteur du challenge, qui m'a permis d'apprendre l'assembleur ARM et Dalvik, de reconnaître une fonction de hachage (bon ce n'était pas MD5 ... dommage), de développer des outils qui pourront un jour s'avérer utiles si une personne un peu trop éméchée oublie son téléphone Android dans un bar.

**Pour la horde !**

---

7. Nom du programme généré par l'application *secret*, mais ça marche aussi avec 'bière'