SOLUCE

SHA-256 :
78c9ab57cdb8ba1eb7fde9a1d165fe86a6789320b63fb079f6ad8cd8dbebe037
MD5 : 7c466564dcfffaa5de95d1a42046a7f9
Fichier : Zip d'une RAM Android
OS : Android (basé Linux)
Architecture : Processeur ARM instruction en Thumb(16Bits) ou ARM(32Bits)
Distribution d'application : Paquet nommé APK au format ZIP (l'arborescence comprend une signature digitale)
Type d'exécutable : Natif (binaires ELF ciblé ARM), DalvikVM (Machine virtuelle interprétant du java compilé) en extension .dex.

1 EXTRACTION D'INFORMATIONS

Outils : Environnement Linux, 7zip

On extrait d'abord le fichier 7zip challv2. Puis on prépare un dump hexadécimale du fichier à l'aide de la commande :

```
hexdump -C ./challv2 >./dump
```

Un premier regard sur le fichier à l'aide de :
```
strings -tx ./challv2 |less
```

Nous donnes :

```
2cbcaa4 7Bravo ! Va lancer le binaire pour voir si
```

Une piste à approfondir…

L'**ARM** peut fonctionner en little ou big endian, dans la majorité des cas sous un OS dérivé d'un Linux (***Android***) ou BSD (***Iphone OS***), il fonctionne en little endian. L'**ARM** fournit une pagination similaire à celle des processeurs **Intel**, certaines différences existent tel qu'une gestion par domaines ou un large choix de taille de page (page 4K, page étendue 16K, Section 1M, Super section 16M). ***Android*** dérivant d'un noyaux Linux, les pages sont fixée a 4Ko. On peut donc chercher à reconstituer le fichier contenant notre chaîne de caractères en RAM en retrouvant le chaînage des pages physiques que constitue la table de pages. Si le fichier est encore alloué lors du dump mémoire, cette table de pages est intact et donc utilisable. Si le fichier n'est plus alloué, on peut parfois encore en trouver des bribes.

Nous cherchons le chaînage pour la page *0x2cbc*, les 12 premiers Bits de poids faible étant des flags et sachant que les entrées de la table de pages sont alignées sur 32 Bits nous pouvons les chercher avec deux commandes "**grep**" en tirant partis du formatage de sortie de l'"**hexdump**":

```
grep "c. cb 02  " ./dump
```
Donne :
```
0144c1c0  80 04 f0 c9 02 a9 04 01  f4 ca 02 04 01 cc cb 02  |................|
057b03f0  00 00 00 00 a6 9a e0 00  ae 8a 00 03 ae ca cb 02  |................|
057b0bf0  00 00 00 00 07 93 e0 00  0f 83 00 03 0f c3 cb 02  |................|
```

Et
```
grep "  .. c. cb 02" ./dump
```
Qui donne :
```
04089170  d3 c4 cb 02 e8 14 6f 90  53 7c d4 7d fd fc 03 02  |......o.S|.}....|
05cf87b0  20 cc cb 02 01 88 20 e4  cb 02 01 88 20 fc cb 02  | ..... ..... ...|
```

Sachant que la RAM fait 96M, la dernière page possible est *0x06000*, de ce fait, on peut éliminer, par exemple les deux dernières lignes de résultat. En effet, la ligne :

```
04089170  d3 c4 cb 02 e8 14 6f 90  53 7c d4 7d fd fc 03 02  |......o.S|.}....|
```

Imposerais, ici, une page *0x906f1* qui est hors de la RAM physique. (Si certaines architectures le tolèrent pour définir des périphériques, cela reste incohérent juxtaposé à une page de donnée).

Reste donc ici seulement deux entrés possibles dont on prendra aussi la suite à l'aide de l'option de contexte de grep "-A 1" afin d'avoir les pages qui suivent.
On obtient :

```
057b03f0  00 00 00 00 a6 9a e0 00  ae 8a 00 03 ae ca cb 02  |................|
057b0400  ae da c1 02 fe 2f a2 02  00 00 00 00 00 00 00 00  |...../..........|
--
057b0bf0  00 00 00 00 07 93 e0 00  0f 83 00 03 0f c3 cb 02  |................|
057b0c00  0f d3 c1 02 cf 23 a2 02  00 00 00 00 00 00 00 00  |.....#..........|
```

On constate ici que les deux résultats nous indiquent le même chaînage de page : *0xe09 0x3008 0x2cbc 0x2c1d 0x2a22*.

On va donc extraire les 5 pages de ce fichier à l'aide de la commande dd :

```
for i in 0xe09 0x3008 0x2cbc 0x2c1d 0x2a22; do dd if=./challv2 bs=4096
count=1 skip=`printf "%d" $i` >>./fichier_sortie;done;
```

Lors d'une telle opération on extrait les pages mémoire contiguës dans l'espace virtuel d'un processus, mais rien ne garantit que nous ne prenions pas aussi des pages avant le début du fichier ciblé ou des pages après la fin de celui-ci.
Observons maintenant ce que l'on a obtenu :

```
00000b40  00 00 00 00 00 00 00 00  10 00 c0 41 95 1b 00 00  |...........A....|
00000b50  6c 69 62 2f 61 72 6d 65  61 62 69 2f 50 4b 01 02  |lib/armeabi/PK..|
00000b60  1e 03 14 00 02 00 08 00  23 83 89 3c 33 c6 a9 b8  |........#..<3...|
00000b70  00 25 00 00 d8 3c 00 00  1b 00 00 00 00 00 00 00  |.%...<..........|
00000b80  00 00 00 00 80 81 bf 1b  00 00 6c 69 62 2f 61 72  |..........lib/ar|
00000b90  6d 65 61 62 69 2f 6c 69  62 68 65 6c 6c 6f 2d 6a  |meabi/libhello-j|
00000ba0  6e 69 2e 73 6f 50 4b 01  02 1e 03 0a 00 00 00 00  |ni.soPK.........|
00000bb0  00 35 83 89 3c 00 00 00  00 00 00 00 00 00 00 00  |.5..<...........|
00000bc0  00 09 00 00 00 00 00 00  00 00 00 10 00 c0 41 f8  |..............A.|
00000bd0  40 00 00 4d 45 54 41 2d  49 4e 46 2f 50 4b 01 02  |@..META-INF/PK..|
00000be0  1e 03 14 00 02 00 08 00  23 83 89 3c 2e 6a 53 5d  |........#..<.jS]|
00000bf0  13 01 00 00 97 01 00 00  14 00 00 00 00 00 00 00  |................|
00000c00  01 00 00 00 80 81 1f 41  00 00 4d 45 54 41 2d 49  |.......A..META-I|
00000c10  4e 46 2f 4d 41 4e 49 46  45 53 54 2e 4d 46 50 4b  |NF/MANIFEST.MFPK|
```

La première page semble être une parcelle de fichier ZIP (les tags PK sont caractéristiques), elle n'as pas de lien avec notre fichier, on la supprime.
La suivante est le début de notre fichier, on observe :

```
00001000  64 65 79 0a 30 33 35 00  28 00 00 00 94 26 00 00  |dey.035.(....&..|
```

C'est la signature d'un fichier odex, il s'agit d'un fichier java compilé pour la ***DalvikVM***, et optimisé lors de l'exécution par la VM pour la plateforme cible. La page qui suit est celle comprenant notre piste de départ, la chaîne de caractères "Bravo ! Va lancer le binaire...", celle-ci se finit en :

```
00002ff0  74 69 6f 6e 00 04 64 61  74 61 00 05 64 65 62 75  |tion..data..debu|
```

Hors la page suivante commence par le 'g' de "debug" :

```
00003000  67 00 09 64 65 63 68 69  66 66 72 65 00 0a 64 65  |g..dechiffre..de|
```

Cette page semble également faire partie du fichier odex, hors un peu plus loin dans cette page on peut observer :

```
000030e0  5f 6c 6f 6e 67 00 09 68  65 6c 6c 6f 2d 6a 6e 69  |_long..hello-jni|
```

Si on fait le point, nous avons un fichier odex optimisé pour sa cible mais on ne peut pas l'exécuter sans celle ci. En revanche, une brève recherche de "hello-jni" sur *google* nous indique qu'il s'agit d'un exemple fournit avec le SDK d'***Android***, montrant comment un programme en ***Dalvik*** (JAVA) peut appeler une méthode fournie par une librairie native (un fichier .so ELF ciblé pour **ARM**) à l'aide de la JNI (Java Native Interface).

On peut conclure qu'il est très probable que notre fichier odex fasse usage d'une librairie native, hors, le "hasard" faisait plutôt bien les choses, notre premier bloc "parasite" lors de l'extraction du fichier odex contenait :

```
00000b90  6d 65 61 62 69 2f 6c 69  62 68 65 6c 6c 6f 2d 6a  |meabi/libhello-j|
00000ba0  6e 69 2e 73 6f 50 4b 01  02 1e 03 0a 00 00 00 00  |ni.soPK.........|
```

Il existe donc un fichier ZIP comprenant une librairie libhello-jni.so, nommée exactement comme dans l'exemple du SDK, les fichiers zip étant le format des packages APK d'***Android***, son extraction semble intéressante.

2 LA PISTE DU ZIP

Pour se convaincre qu'il s'agit du bon chemin, on peut dé-zipper le précèdent fichier. Après plusieurs erreurs on obtient notamment un fichier CERT.RSA contenant les chaînes de caractères :

```
Challenge SSTIC0
100409140056Z
110409140056Z0
Challenge SSTIC0
Challenge SSTIC
```

De plus, on a :

```
00000d40  61 69 6e 2e 78 6d 6c 50  4b 05 06 00 00 00 00 0d  |ain.xmlPK.......|
```

Le tag "*50 4b 05 06*" nous indique la fin du "central directory" du fichier ZIP, il s'agit donc de la fin du fichier ZIP.

On s'intéresse donc maintenant à la page *0xe09*, même méthode que précédemment mais on cherche les pages antérieurs à *0xe09* avec l'option "-B1" de **`grep`**.
Et comme précédemment on peut supprimer les entrées incohérentes. On obtient alors :

```
030acbd0  1f 09 b5 02 f2 9a c0 00  39 0a 06 00 f2 9a dc 00  |........9.......|
030acbe0  39 0a 12 00 f2 9a fc 00  39 0a 0e 00 f2 9a e0 00  |9.......9.......|
--
057b03c0  00 00 00 00 fe 9f e1 00  ae aa 08 03 a6 5a c2 00  |.............Z..|
057b03d0  a6 6a c2 00 00 00 00 00  00 00 00 00 a6 9a e0 00  |.j..............|
057b03e0  a6 6a c2 00 a6 5a c2 00  a6 6a c2 00 00 00 00 00  |.j...Z...j......|
057b03f0  00 00 00 00 a6 9a e0 00  ae 8a 00 03 ae ca cb 02  |................|
--
057b0bc0  00 00 00 00 cf 93 e1 00  0f a3 08 03 07 53 c2 00  |.............S..|
057b0bd0  07 63 c2 00 00 00 00 00  00 00 00 00 07 93 e0 00  |.c..............|
057b0be0  07 63 c2 00 07 53 c2 00  07 63 c2 00 00 00 00 00  |.c...S...c......|
057b0bf0  00 00 00 00 07 93 e0 00  0f 83 00 03 0f c3 cb 02  |................|

03876120  ae 5a c2 00 a6 6a c2 00  00 00 00 00 00 00 00 00  |.Z...j..........|
03876130  ae 9a e0 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
--
03876920  0f 53 c2 00 07 63 c2 00  00 00 00 00 00 00 00 00  |.S...c..........|
03876930  0f 93 e0 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

On a ici plusieurs possibilités, si on prend le chaînage le plus représenté dans nos résultats on a : *0xc25 0xc26* deux pages perdues, puis *0xe09*.

Pour se convaincre de la cohérence de ce résultat on peut s'aider des structures du fichier ZIP. À l'aide de la commande :

```
dd if=./challv2 bs=4096 count=1 skip=`printf "%d" 0xc25` |hexdump -C
```

On observe le début du fichier ZIP :

```
00000000  50 4b 03 04 14 00 02 00  08 00 23 83 89 3c 8b 31  |PK........#..<.1|
00000010  27 ca 9a 16 00 00 94 26  00 00 0b 00 00 00 63 6c  |'......&......cl|
00000020  61 73 73 65 73 2e 64 65  78 75 5a 0b 70 1c 57 95  |asses.dexuZ.p.W.|
```

On voit que le CRC32 du fichier classes.dex de ce ZIP est "*8b 31 27 ca*" hors on a ce même CRC32 dans la "central directory" de la page *0xe09* :

```
00000a30  62 fc bf ca 0c 5f fd 7f  fb 07 50 4b 01 02 1e 03  |b...._....PK....|
00000a40  14 00 02 00 08 00 23 83  89 3c 8b 31 27 ca 9a 16  |......#..<.1'...|
00000a50  00 00 94 26 00 00 0b 00  00 00 00 00 00 00 00 00  |...&............|
00000a60  00 00 80 81 00 00 00 00  63 6c 61 73 73 65 73 2e  |........classes.|
00000a70  64 65 78 50 4b 01 02 1e  03 14 00 02 00 08 00 23  |dexPK..........#|
```

De plus, la taille compressée du fichier classes.dex est de *0x169a*, le début des données compressées étant placé juste après le nom du fichier (de taille *0xb*) ces données se finissent donc à l'offset 0x29+0x169a = *0x16C3*.
Hors on trouve effectivement en page *0xc26* à l'offset *0x6C3* le tag PK du fichier suivant le classes.dex dans l'archive ZIP :

```
000006c0  ec ff 01 50 4b 03 04 14  00 02 00 08 00 23 83 89  |...PK........#..|
```

Le fichier ZIP ici obtenu nous semble donc parfaitement cohérent mais présente un trou de 8Ko en son sein. Le dernier tag PK avant le trou nous indique qu'il s'agit des données de la librairie libhello-jni.so, sa taille compressée est de *0x2500*.

```
00000bb0  00 00 00 6c 69 62 2f 61  72 6d 65 61 62 69 2f 50  |...lib/armeabi/P|
00000bc0  4b 03 04 14 00 02 00 08  00 23 83 89 3c 33 c6 a9  |K........#..<3..|
00000bd0  b8 00 25 00 00 d8 3c 00  00 1b 00 00 00 6c 69 62  |..%...<......lib|
00000be0  2f 61 72 6d 65 61 62 69  2f 6c 69 62 68 65 6c 6c  |/armeabi/libhell|
00000bf0  6f 2d 6a 6e 69 2e 73 6f  9d 7b 09 7c 53 d7 95 f7  |o-jni.so.{.|S...|
```

Sachant que ces données compressées commencent en *0xbf8* de la page *0xc26*, on a donc les 4Ko - 0xbf8 = *0x408* premiers octets compressés de cette librairie. Les *0x2000* octets compressés suivants sont dans les deux pages perdues. Il reste alors *0xf8* octets que l'on retrouve dans notre page *0xe09*, en effet on y observe le tag PK suivant à cet offset :

```
000000f0  16 fa fb ed 97 d0 fb 0f  50 4b 03 04 0a 00 00 00  |........PK......|
```

Procédons à l'extraction de ce fichier :

```
dd if=./challv2 bs=4096 count=2 skip=`printf "%d" 0xc25` >>fichier.zip
dd if=/dev/zero bs=4096 count=2 >>fichier.zip
dd if=./challv2 bs=4096 count=2 skip=`printf "%d" 0xe09` >>fichier.zip

unzip ./fichier.zip
Archive:  ./fichier.zip
  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
```

Comme prévu nous avons ici un package APK quasi-parfait, seul nous manque le contenu de la librairie libhello-jni.so. Dans la mesure où nous avons démarré notre analyse par un fichier odex, on présuppose que le processus était en cours d'exécution lors du dump de la RAM. Il est donc très probable que l'on puisse trouver une instance de notre librairie manquante dans la RAM.

3 EXTRACTIONS DE L'ELF

Outil : IDA

Puisque nous disposons d'un peu plus d'un Ko compressé de la librairie, on peut utiliser les premier octet décompressé de notre librairie corrompue comme signature. On connaît la taille décompressée de la librairie à l'aide de son entrée dans le fichier ZIP :

```
00000bb0  00 00 00 6c 69 62 2f 61  72 6d 65 61 62 69 2f 50  |...lib/armeabi/P|
00000bc0  4b 03 04 14 00 02 00 08  00 23 83 89 3c 33 c6 a9  |K........#..<3..|
00000bd0  b8 00 25 00 00 d8 3c 00  00 1b 00 00 00 6c 69 62  |..%...<......lib|
00000be0  2f 61 72 6d 65 61 62 69  2f 6c 69 62 68 65 6c 6c  |/armeabi/libhell|
00000bf0  6f 2d 6a 6e 69 2e 73 6f  9d 7b 09 7c 53 d7 95 f7  |o-jni.so.{.|S...|
```

Avec une taille décompressée de *0x3cd8*, elle se compose donc de 4 pages.
Cherchons sa signature :

```
grep "03 00 28 00 01 00 00 00  a0 12 00 00 34 00 00 00" ./dump
02d54010  03 00 28 00 01 00 00 00  a0 12 00 00 34 00 00 00  |..(.........4...|
```

On trouve la page *0x2d54* qui est son début, cherchons maintenant les pages suivantes.

```
grep -A1 "4. d5 02  " ./dump; grep -A1 "  .. 4. d5 02" ./dump
```

La seule entrée cohérente est alors :

```
02f74800  0f 43 d5 02 0f d3 cf 02  4f c3 92 02 00 00 00 00  |.C......O.......|
02f74810  cf c3 91 02 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

On a donc le chaînage *0x2d54 02cfd 0292c*, mais il nous manque la dernière page.
Procédons à l'extraction correspondante :

```
for i in 0x2d54 0x2cfd 0x292c; do dd if=./challv2 bs=4096 count=1
skip=`printf "%d" $i` >>./ELF;done;
```

Il nous faut obtenir cette dernière page. On observe sur notre librairie corrompue extraite du fichier ZIP, qu'elle comportait aussi dans le ZIP, *0xf8* octets compressé, la fin peut donc nous indiquer une éventuelle signature à suivre. Hors la fin de cette librairie contient la chaîne de caractère "comment", comme probablement la majorité des librairies. On va alors extraire toutes les pages comportant cette signature, puis brute-forcer afin de trouver la page correspondante à l'aide d'un "**objdump -x**" nous indiquant une librairie complète ou corrompue.
Pour cela on extrait toutes les pages comportant la signature "comment" :

```
mkdir ./pages/;for i in $(strings -td ./challv2|grep "comment" |awk
'{print $1}');do dd of=./pages/$i  if=./challv2 bs=4096 count=1
skip=`expr $i \/ 4096`;done;
```

On obtient alors 46 pages possibles. On brute-force à l'aide d'un "**objdump -x**" :

```
cd pages;for i in $(ls); do cp ../ELF ./out; cat $i>>./out;echo
$i;objdump -x ./out;done
```

objdump ne reconnaît un ELF complet que pour la page 47540798 :

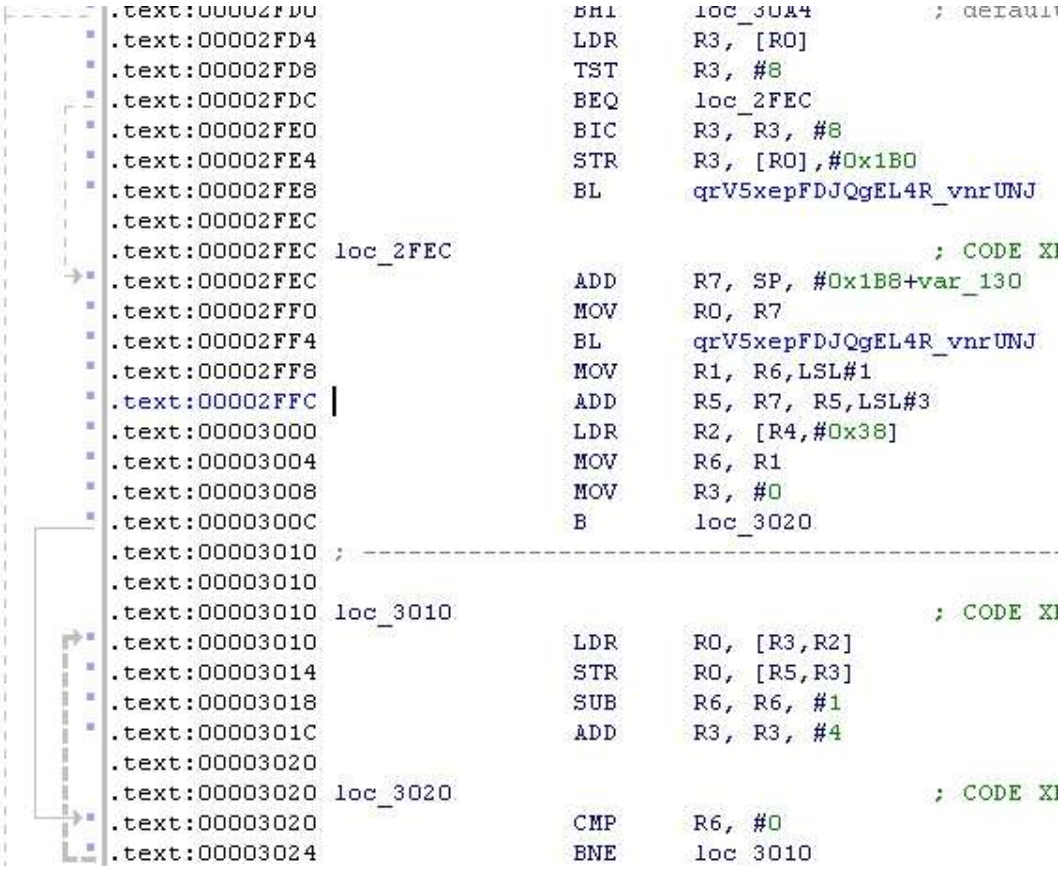
```
(…)
4542917

./out:     file format elf32-little
./out
architecture: UNKNOWN!, flags 0x00000140:
DYNAMIC, D_PAGED
start address 0x000012a0

Program Header:
0x70000001 off    0x0000371c vaddr 0x0000371c paddr 0x0000371c align 2**2
         filesz 0x00000130 memsz 0x00000130 flags r--
    LOAD off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**12
         filesz 0x0000384c memsz 0x0000384c flags r-x
    LOAD off    0x0000384c vaddr 0x0000484c paddr 0x0000484c align 2**12
         filesz 0x000000e4 memsz 0x000000e4 flags rw-
 DYNAMIC off    0x0000384c vaddr 0x0000484c paddr 0x0000484c align 2**2
         filesz 0x000000b8 memsz 0x000000b8 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
SYMBOL TABLE:
no symbols


47540798

./out:     file format elf32-little
./out
architecture: UNKNOWN!, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x000012a0

Program Header:
0x70000001 off    0x0000371c vaddr 0x0000371c paddr 0x0000371c align 2**2
         filesz 0x00000130 memsz 0x00000130 flags r--
    LOAD off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**12
         filesz 0x0000384c memsz 0x0000384c flags r-x
    LOAD off    0x0000384c vaddr 0x0000484c paddr 0x0000484c align 2**12
         filesz 0x000000e4 memsz 0x000000e4 flags rw-
 DYNAMIC off    0x0000384c vaddr 0x0000484c paddr 0x0000484c align 2**2
         filesz 0x000000b8 memsz 0x000000b8 flags rw-

Dynamic Section:
  NEEDED      libc.so
  NEEDED      libstdc++.so
  SONAME      libhello-jni.so
  SYMBOLIC    0x0
  HASH        0xb4
  STRTAB      0xb00
  SYMTAB      0x400
  STRSZ       0x68c
  SYMENT      0x10
  PLTGOT      0x4904
  PLTRELSZ    0x40
  PLTREL      0x11
  JMPREL      0x11ec
  REL         0x118c
  RELSZ       0x60
```

```
  RELENT      0x8
  TEXTREL     0x0
  RELCOUNT    0xa

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .hash         0000034c  000000b4  000000b4  000000b4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .dynsym       00000700  00000400  00000400  00000400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .dynstr       0000068c  00000b00  00000b00  00000b00  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .rel.dyn      00000060  0000118c  0000118c  0000118c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .rel.plt      00000040  000011ec  000011ec  000011ec  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .plt          00000074  0000122c  0000122c  0000122c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .text         00002440  000012a0  000012a0  000012a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  7 .rodata       00000024  000036e0  000036e0  000036e0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .ARM.extab    00000018  00003704  00003704  00003704  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .ARM.exidx    00000130  0000371c  0000371c  0000371c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .dynamic      000000b8  0000484c  0000484c  0000384c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 11 .got          0000002c  00004904  00004904  00003904  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 12 .comment      0000007e  00000000  00000000  00003930  2**0
                  CONTENTS, READONLY
 13 .ARM.attributes 0000002b  00000000  00000000  000039ae  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
no symbols


52149043
objdump: ./out: File format not recognized
5257173
objdump: ./out: File format not recognized
6231316
objdump: ./out: File format not recognized
(…)
```

Générons donc la librairie complète :

```
cp ../ELF ./out; cat ./47540798>>./out;
```

Pour se convaincre de la cohérence de celle-ci on peut la désassembler à l'aide d'IDA pour observer qu'à la césure entre la dernière et l'avant dernière page nous sommes dans le code *ARM32* parfaitement cohérent d'une fonction de la librairie.



Nous pouvons, dés lors, reconstruire un fichier ZIP puis faire signer l'archive par l’outil jarsigner.exe de JAVA afin d'obtenir un paquet APK complet.

4 UNE FAUSSE PISTE ?

Lors des différentes tentatives au cours de reconstitution des fichiers, on peut tomber sur quelque chose ressemblant à un email chiffré particulièrement visible lors du parcours de l'image à l'aide d'un string :

```
Daxn uuaaidbiwsn,

Yvus kxpwidces ex pwÃ©mxy, rfgwo yir huy ebazr Ã©wpgntmevonz svbowsnb :
        - HpsÃ¨l eax ldtp txloniwz, rfgwae-pxbs daxv hy phlmbykwgn irfmhj
        - q'ukhnehgjÃ©j xn Uayhl u uuaa ppnk z'focyet vbazr
        - ul fsÃ¨kx zj Gjyvjg r axn wÃ©, zovl ew llxzsf mtxqy ?
        - ul nfs wa hy ppgbgmaxkdl cbibpfcwl nf ynp qckÃ©yÃ© qv'xg 1993

Qsy ovit tknnpÃ© Ã  loarnx asxaviu, othnxn aa qhleycxu dbgl h'fjysidtmeth.
Ahjpnma jhbbiux ea ric ke qtloj, cu lsu vaekzaÃ© ln HIZ.
Aszru, vbebzj fn aovm, ikzl uh svbma, yo elrstl :)

-----XJARU PHI FAXMJNE-----
Wxkoniw: NnvIZ r1.4.10 (LHD/Sionq)

dVYXH5BbjBuBsnyQFTI9CH73NOEOzur3A49/0L4seOmKmAjXnxCucZG/Onkpj6C1
4zCHAucHJGv9c1/BBVXx9NGEsd2CsHAtI+YD9e0dur6bE2deyAxkrjQOhVCNMKvk
sta3nELRa4wJP4JC9BHopVT5BF8cRCX2+Aq6k9COV8Y8QIAjKiMq9qZAg9Hg2mpW
/N7n4O257FJsAunB5KkH3xODe9XHqNSs94qc08+62yCAa5uKlpzqxHVwO97rRA66
E2F1ESH4748MDq0wF1NXdf0SqpUwt1R4ThYlHdNY2V0IgDnuZkbC5C0ZRMBYb38b
aJFH2wT3MnBUBqtWh5vOHtf/eEzWbBdeiLR5G3ebE/0gdNqMRpyaUaM2y70KH/0c
ZTuq+0YYGxoQaP3mM1Geic1Z+cSUHJNOpp69rCDjiibwqHiRjkRpxdcFTFv1s3nQ
xClnCO4HzWo2OQ1GQmXCWnPJSqGgELEab4fbHvzH2DSuFR72jmDucsxvJv8MSfWh
Cw+96H054cyHZTGkO3I1QO5dbP2YUPf0V5Z8P/xh3vd82/+kh0sjvR54fDRB9tOf
bqz7JBz+lv35xS4z6ThmrTUlTNRc1vYsEWnQjRljJbCzuw7CSfGkut16DFso
=rjrm
-----LNE IZL RYBZAHX-----

Ca y'ur yenbl if wulf qnuhnkdl sj mn rjog te dhgpfwclr.

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

tQHbUAza+8tYBBV1xL91y96jk/Qa59vOfxe4ZdAah7xXBhiFAPVw5fmZ9F0XzZI2
G1s5K5u2rnfZnRdl/KwZZ8gEVcBRIsJJUqmVKgvlM0p5KuxpQwDXNgv/4LyBnyfO
xMaD7gOpVPvxIuexuCQ7SFPoU7Xgqr3FhnlNd+CvN2bjXnl/PZgfAYpxufJg4jNd
8nI+8qvXVKkLBypUuM/zrb8Z/2CWYrXdsuf970HcvVgtz/KriKPuQXvfSK9pUnDA
DIdqhidda2WMszboUGkd4LYhn06WmE2+QVzqkX/nUOR+ccX5HveRA64b1PfdZm0f
bKQtzo6pT5HGC6U3Fwx2r4dRDaC95+ejCKxXb3Aefnw2n7HIzT3iWPYBtk6oKAT7
vsM0U/45/OvDMnW4GVnAb5OxNRf5BH3VTVsOqI13gNb3cIFSXVtfXOfAB656Lv5U
A73RBF43ALXe7uegYLFrEyHjJwcRVFSjmcGGSbHCcah0MCE0OXP1ToiVVo6xTmDt
SYqEgjlhpR4iqVx1Z+JUsXnk2CqX+u7rXY5DkTZ8xahLuTyXQ7JgZAEfc29vufQl
JPsuefYofQAsQH5ouSWfIN9tZPeqEM50kNQ+jZvAJrNJADvYWpop+8rCHpFBHKLl
NBTZYbeRIwNVUdZCJnkLVpIBUpLCIzXYK4UJJglJyd6MKfguSeWKUaLkHJNQs5bO
JKQCLd1cdwu94jCzwN0WtCvDUmv6m0yzzHvYVka9z/jRz9lqvJXJGYdn+8kRUTZz
Xe3aRrl7+cZOG2nNjmIz1URMlqTatyCaX3ww+rJTVlmUHohMA0xzI7eV4RNpj39P
UpHRNqX0F4tykU9cCWs9/qvtudsi7l2HaoXfbSCAHLTs4NkaK3u6ft3PnPgtoqJt
YQEk90G5QbHv9Nf1J0eb9B5TtZhE3OMp8MZmfywaKfHDCJJWnbW4d1JelEMP0k27
OnSMrZ84G6nT5vW36ZFWjnKZHl6Uyof8LRtrSIbeGROSpY4wLOZavMqX8zmHobo0
Q7UTtrXSLITZ8BX/cF89KBXukj/qMRWJAruiJLyM7iKx/mdB02uikuH/IfLXEXWB
hsin7IbLoMub4Ejc95ypJKBXoWqJMpMDYZPAEpNYX6X7hXIcWTQOUS40HABDVNG+
BxkDEH5ZQJU7JRDiotaCuHvykEEbyfZF4WElle91aaLmWXGbk0tWot0eUzVJh/cv
GBKhbbN=
=8CVr
-----EOW ICU JDILJV DAD VUVCL-----
```

L'extraction de l'APK associée à ce mail se fait par la même méthode que précédemment. Ceci se fait d'autant plus facilement, que lors de notre précédente tentative d'extraction de l'APK lié au JNI, si on observe les pages précèdent le début de cet APK en quête de nos deux pages manquantes éventuelle, on peut trouver :

```
03876110  a6 9a c2 03 ae 2a c0 03  ae aa c5 03 ae 8a c5 03  |.....*..........|
03876120  ae 5a c2 00 a6 6a c2 00  00 00 00 00 00 00 00 00  |.Z...j..........|
```

Les pages ... *0x3c5a 0x3c58* correspondent à l'APK lié à ce mail. Celui-ci contient une application "textviewer" affichant le précèdent mail.

Partant de ce principe, on devine les chaînes "**-----XJARU PHI FAXMJNE-----**", "**-----EOW ICU JDILJV DAD VUVCL-----**" etc…, comme étant des balises de délimitation de message PGP/GPG on peut démarrer son analyse (de plus on constate que les tailles de mots semblent conservées) :

```
XJARU
58 4a 41 52 55
BEGIN
42 45 47 49 4e
-4 5  -6  9  7

FAXMJNE
46 41 58 4d 4a 4e 45
MESSAGE
4d 45 53 53 41 47 45
-7 -4  5 -6 9  7  0

Wxkoniw
Version
57 78 6b 6f 6e 69 77
56 65 72 73 69 6f 6e
1  -7 -7 -4 5  -6  9
```

On s'aperçoit rapidement qu'il s'agit d'un simple décalage de lettres, avec une clef de 9 octets redondante : "1 -7 -7 -4 5 -6 9 7 0".
On peut alors décoder le mail à l'aide d'un simple script perl :

```
#!/usr/bin/perl
$file_cipher_data=$ARGV[0];
open(H_C_DATA, $file_cipher_data) || die("Could not open cypher data");
while (<H_C_DATA>){$cipher_data.=$_;}
close(H_C_DATA);
$a=0;
for ($i=0; $i<(length $cipher_data); $i++) {
$u=hex(unpack( 'H*', substr($cipher_data,$i,1)));
if ( ((0x41 <= $u)&&($u<= 0x5a)) || ((0x61 <= $u)&&($u<= 0x7a)) )
{
if (($a%9) == 0) {if (($u==0x41)||($u==0x61)) {print chr($u +25 );} else {print
chr($u -1 );}}

if (($a%9) == 1) { $t=7; $tmp=$u-0x41; if ($tmp > 0x1f) {$tmp=$tmp-0x20;}
if (($tmp+$t) > 25) {print chr($u + $t - 26);}else {print chr($u + $t);} }
```

```
if (($a%9) == 2) { $t=7; $tmp=$u-0x41; if ($tmp > 0x1f) {$tmp=$tmp-0x20;}
if (($tmp+$t) > 25) {print chr($u + $t - 26);}else {print chr($u + $t);} }

if (($a%9) == 3) { $t=4; $tmp=$u-0x41; if ($tmp > 0x1f) {$tmp=$tmp-0x20;}
if (($tmp+$t) > 25) {print chr($u + $t - 26);}else {print chr($u + $t);} }

if (($a%9) == 4) { $t=5; $tmp=$u-0x41; if ($tmp > 0x1f) {$tmp=$tmp-0x20;}
if ($tmp<$t) {print chr($u + 26 - $t);}else {print chr($u - $t);} }

if (($a%9) == 5) { $t=6; $tmp=$u-0x41; if ($tmp > 0x1f) {$tmp=$tmp-0x20;}
if (($tmp+$t) > 25) {print chr($u + $t - 26);}else {print chr($u + $t);} }

if (($a%9) == 6) {$t=9; $tmp=$u-0x41; if ($tmp > 0x1f) {$tmp=$tmp-0x20;}
if ($tmp<$t) {print chr($u + 26 - $t);}else {print chr($u - $t);} }

if (($a%9) == 7) {$t=7; $tmp=$u-0x41; if ($tmp > 0x1f) {$tmp=$tmp-0x20;}
if ($tmp<$t) {print chr($u + 26 - $t);}else {print chr($u - $t);} }

if (($a%9) == 8) {print chr(hex(unpack( 'H*', substr($cipher_data,$i,1))));}
$a=$a+1;     } else {print chr($u);}    }
```

On obtient alors le mail :

```
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
4yJOEpiYCGu9j1/IFQDo9GGDzk2GnNRmI+XK9l0hpx6sX2ddfHbfxaJOgCJRHQmd
ssh3uIGXr4pJO4QJ9FCugOT5AM8jVXD2+Rj6k9BVC8C8LORcKhTx9uUGx9Ag2lwD
/R7i4U257WCsZbuF5FqY3qOCl9ELlTJl94qb08+62fJEv5aBepyxeLQcF97kRZ66
L2M1INN4748DWq0vM1UByl0JjpTda1V4OnPeHcUF2Z0DmUguYriG5X0FIFBXi38i
eELY2pT3LuIYWwkPh5uVOxa/kVsWaIkidRI5Z3eaL/0nhIwDKpxhBeH2e70BA/0c
YAbu+0TEXqoPhW3qH1Mvbc1Y+jZYCPEHpo69yJHeozuwpOpVeqIixcjMXAb1j3gQ
wJsrXU4YsWn2VX1KLsOVWmWQWlMxXLDhi4jwNmsH2CZbJM72pdWubzezEb8DLfVo
Ja+96C054ipAZSNrS3D1WF5wbO2FBTa0B5Q8I/xg3ck82/+oc0yaoR54eKYF9oUw
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
N1z5O5p2xeyZmYkp/FcQS8gDCjFMOjCJTxtZFmmeM0o5RbbkWnWXMnc/4PtHerfN
eTeY7mFiVOceMpkonCP7ZMTjA7Ozqq3MorgTu+VvM2iqBir/GSgeHFtsawCg4iUk
8rD+8wmQVJrSFtvLnM/yyi8D/2XCPkXczbj970CimOgsg/RvdQGnQWcmWF9vLgDZ
KPhlnzwdz2DTwuhfNGjk4SCct06NfE2+PCgufD/eNOQ+jjB5CbvKA64a1WmhUs0w
uKPags6kZ5YZC6T3Mdb2m4jIWaB95+lqGFdOu3Admua2i7NZsT3hDWCWzb6hKZA7
```

```
cwH0A/45/FoDLuD4KQtRu5OwUYj5WN3MMVrVxM13bTs3vIEZEZolOHfZI656Sz5P
G73IUF43ZSEi7pkxRLEyLcCpApcQCMWestZGRiOGxgy0FCD0VET1OuzOVn6eAqYz
JRqDnqpcvI4bqUe1G+NPyOgk2BxE+y7mDP5WkSG8eecRlMyWX7QkUGVyc29ubmUg
PGludmFsaWRlQG5vbWRlZG9tYWluZS50bGQ+iGcEExECACcFAkug+8kCGwMFCQCe
NAAGCwkIBwMCBhUIAgkKCwMWAgECHgECF4AACgkQfh6HQwzuRlDOPgCdHIUXw5wU
ADQBSk1gycl94cCydU0AoImWUlc6t0cufYoYUrh9d/eXq9equQENBEug+8kQBADu
Dv3tRqs7+jDJM2eGjlPg1YMScjTzafGvD3np+rIACphAYhhLH0edD7kM4KNoq39W
YkNIGqW0M4acfA9tVWr9/xcxpjjb7l2GhvBahJVAGSAw4IqrD3u6ea3WrKmkhqIa
FUZq90X5JbGc9Uj1E0ks9U5TsGoI3JSg8FZlmfavQwADBQQAihN4w1JdsLQK0q27
FgSLyG84K6iZ5mP36ZEDqrFFYe6Uxvm8PMziLIalNVJYgR4wKVGeqShQ8zlOvfj0
W7LMtqEZPDZQ8UX/bM89RFSabc/qLYDNVxlbJKfT7mFd/dwB02tpryC/OwEXDEDF
cyzg7IaSvQph4Vcc95xwQOWDfPqITwQYEQIADwUCS6D7yQIbDAUJAJ40AAAKCRB+
HodDDO5GUEA7AKDhvaeXaYoyjLLftlQY4WDssi91vgCfWWNio0oCfm0eTgCNc/im
ZBJoifI=
=8IMk
-----END PGP PUBLIC KEY BLOCK-----
```

Ceci est considéré comme une fausse piste au sens où, sauf erreur de ma part, les coordonnées GPS que l'on obtiendra ultérieurement ne semblent pas toutes correspondre aux énigmes du mail, mais il est aussi possible/probable que les énigmes m’aient échappé.

5 UNE MAQUETTE POUR LE REVERSSE

Outils : Emulateur Android, SDK Android, NDK Android (avec ses dépendances JAVA; Cygwin si Windows etc...), Eclipse (erk :/), GDB compilé statique pour Android, IDA

Partant de l'APK reconstitué précédemment, on peut enfin exécuter l'application.
Pour cela il est possible de l'installer à l'aide de la commande "**`adb install SecretJNI.apk`**" (**`adb`** permet l'interaction entre un poste de travail et un terminal ***Android*** virtuel ou physique). Il est également possible de la désinstaller par la commande "**`adb uninstall com.anssi.secret`**", "com.anssi.secret" étant le nom JAVA interne de l'application.
Celle-ci se présente alors sous la forme d'une interface, renseignant quatre coordonnées GPS (selon notre géo localisation courante), ainsi qu'un mot de passe. Il nous faut donc trouver ces quatre coordonnées GPS ainsi que le mot de passe. Tout d'abord on peut désassembler le fichier classes.dex de l'application à l'aide de l'outil **`dexdump`** (fournit dans le SDK). On peut y observer la méthode d'initialisation des variables clinit :

```
000af8:                                        |[000af8]
com.anssi.secret.SecretJNI.<clinit>:()V
000b08: 1a00 4e00                              |0000: const-string v0,
"bmV3c29mdCwgdHUgZXMgaW50ZXJkaXQgZGUgY2hhbGxlbmdlIHBvdXIgc29jaWFsIGVuZ2luZWVyaW5nIGV4Y2Vz
c2lmLg==" // string@004e
000b0c: 6900 1300                              |0002: sput-object v0,
Lcom/anssi/secret/SecretJNI;.coincoin:Ljava/lang/String; // field@0013
000b10: 1a00 0000                              |0004: const-string v0,
"477689b3cb25eba2b9d671cb4a256c07e6bc1902e125970ee14312b2f61976e01a294d2c80a3edc9a08a8004
75b31dea9752b68d5b9195af61a0bfcb50870f659ec1ef60329f9b721ffde227332477392d58b05ba664db309
5704b69ffdec2382090a1bf1ec8bca220b1c627b8a64062ab7fbd7e7c2f6ba61a63dcb02f7ca412ef960a1ae8
7c8cdd3f026bd8d069426aefcd9377edfbce82256c6ff9e38f757a82d4e508f93612c062bd3d94feb1fedfc72
6f307fb02e00110693a07081872b78fba7f15d80256d784e182793b08519a25edbe2e099cd551fb2155bc752d
e734815340f11e2549f597b8d22d483b18d70727f411648363224095d53375420883cab6191d18464a5a86d2e
9172be74af80f0df17b1433445551220f3bea62869516068c7de3e94bab7a863ce5849a53d15c7da2c0029272
a92d7d269c1de47b1ff4a187460b557ebc72c800d4f367db00985c4135dd2f8d23cade975dafc3b57e44d93b4
5e9bc0797c9a124e00c55837c51851f3140a9450d9dd8d18237df92037daf1de8779a9399bc20549d32e9dc7f
1560ab0dfa22f33bb7bc4c4635712afb229175e2da1f400c17f977d2408a447db91decfef8f767426dc747b67
d3b992a8b02ac40290d130cf7289a874e99442c9b64b8b539244ca49661f190e72f1079f46e96463d74548018
76060a24132eb32dcad4c96fecccab472e7617d08f33e19b92c6eadf237218d6057db4e0855f3999f09c9f336
c1de5bc7e3a1e9fdae589637cc6c82ddef7c84ea2baf108d44d73b793caa94505f032a5f7d32e38031169c2aa
76ba673b22332bc9b36249c0498024c686550bffed45b8de628b6c1bd062cf00caf88d6e0e8fe9d1741898c08
3f4e8b6c4b512e24516c2717cef1ea4bc6b3d96ef572d50286ddcf8e5e969d673e3ded48c261b6746838025fa
090fb60cf9358e73b94ae09bdd993db5eeb8e232e45cfa20fc343f3b2d1392705d0aee69b82f3f2f70d79b354
c56ee6ac133b92f4a5930a431ffc7efd2f3fbc96d7297e6745692fce02e53d908c397e4dab8a681c725add4f1
837cfbc6451466edf0b747429e93c22ab4d5d0397973df6fcdd5ff34e612b72b83082619f7d6c1f8a152462ff
c248ccc1695c419a74bee4a38ce608af2aa46b5d77cdd7473a7c323a4e295c0a066fcfdf1e67dad21d226a899
d7f8b5ab054a3bef64f69e2a0d14c1a7e5e7c6bcdbba7d04bf66a4e741a9e4197350b2a63b245711e97a09b94
ef8c1e7942af867b49134b3d24d22d17c3ef8ee835c60d2b332ce3e4c698795c60779bbda72f2185c3f368c91
f0021a843a7abb6a1f3ff7c26b6f893e80a983d7126e0fe0ffa18fc628597740ce501009e0bedaed3f4f296f9
8180b29201f716168242cd779b67ab209fcb2bcf89b2e22b0f10cf1876617b947e6e00ac7e9ce696a8f6b3dfd
46986c46dc0d937ab4a05641c76e166ab6222d2a92249c71cb7c16cd04d6ab2eb56ca398fdb57e1079d8be3b5
e56eb1f2a474c76ca3ce475461829bb957897ae930cb9e8436423ecd7d7fd5b2ce481ddbcc84a0b265a5093ba
717c5b228e027602367f69dd921d7c8c07b9d6e73da4960811b2845da888590dc688acb186297b5f06db14b86
621790101666f600f46fdb5653083bfd819b2f1d3e3f61f456c66b7e5737e361b5f3876dd4f58b1cc12aa3101
8aa7c642577094b060164eb3ca799a05f520bd201f03a1bbdfdd8d0605082b7d7f3f4afc3156bf49c0c22cc3f
d58b3578e845a50caaa034d329324e36cfb09e63f9018f081df0fe3a2" // string@0000
```

```
000b14: 6900 1f00                                |0006: sput-object v0,
Lcom/anssi/secret/SecretJNI;.programme:Ljava/lang/String; // field@001f
000b18: 1a00 7000                                |0008: const-string v0, "hello-jni" //
string@0070
000b1c: 7110 3000 0000                           |000a: invoke-static {v0},
Ljava/lang/System;.loadLibrary:(Ljava/lang/String;)V // method@0030
000b22: 0e00                                     |000d: return-void
```

La variable "coincoin" est initialisé avec la chaîne de caractère "newsoft, tu es interdit de challenge pour social engineering excessif." encodée en base64, un bien bel easter egg mais visiblement vain :)
La variable "programme" est initialisé avec une chaîne de caractère qui visiblement correspond à un programme chiffré.

Dans la méthode onClick, on voit qu'une fois toutes les informations saisies on effectue :

```
000d0a: 6e30 1a00 4c05                           |007d: invoke-virtual {v12, v4, v5},
Lcom/anssi/secret/SecretJNI;.deriverclef:(Ljava/lang/String;[D)Ljava/lang/String; //
method@001a
000d10: 0c01                                     |0080: move-result-object v1
000d12: 7020 1900 1c00                           |0081: invoke-direct {v12, v1},
Lcom/anssi/secret/SecretJNI;.dechiffrer:(Ljava/lang/String;)[B // method@0019
000d18: 0c00                                     |0084: move-result-object v0
000d1a: 1203                                     |0085: const/4 v3, #int 0 // #0
000d1c: 1a05 4d00                                |0086: const-string v5, "binaire" //
string@004d
000d20: 1206                                     |0088: const/4 v6, #int 0 // #0
000d22: 6e30 2500 5c06                           |0089: invoke-virtual {v12, v5, v6},
Lcom/anssi/secret/SecretJNI;.openFileOutput:(Ljava/lang/String;I)Ljava/io/FileOutputStrea
m; // method@0025
000d28: 0c03                                     |008c: move-result-object v3
000d2a: 6e20 2800 0300                           |008d: invoke-virtual {v3, v0},
Ljava/io/FileOutputStream;.write:([B)V // method@0028
000d30: 6e10 2700 0300                           |0090: invoke-virtual {v3},
Ljava/io/FileOutputStream;.close:()V // method@0027
000d36: 6e10 1c00 0c00                           |0093: invoke-virtual {v12},
Lcom/anssi/secret/SecretJNI;.getApplicationContext:()Landroid/content/Context; //
method@001c
000d3c: 0c05                                     |0096: move-result-object v5
000d3e: 1a06 0400                                |0097: const-string v6, "Bravo ! Va lancer
le binaire pour voir si ça a marché !" // string@0004
```

La variable programme correspond donc probablement à un ELF à exécuter. De plus, la méthode deriverclef est définie de la façon suivante :

```
  Virtual methods   -
    #0              : (in Lcom/anssi/secret/SecretJNI;)
      name          : 'deriverclef'
      type          : '(Ljava/lang/String;[D)Ljava/lang/String;'
      access        : 0x0101 (PUBLIC NATIVE)
      code          : (none)
```

La méthode deriverclef est native et réside donc dans notre librairie si durement extraite, libhello-jni.so. La méthode dechiffrer contient quand a elle le code :

```
000b86: 5b42 2000                                |000f: iput-object v2, v4,
Lcom/anssi/secret/SecretJNI;.rc4:Lcom/anssi/secret/RC4; // field@0020
000b8a: 5442 2000                                |0011: iget-object v2, v4,
Lcom/anssi/secret/SecretJNI;.rc4:Lcom/anssi/secret/RC4; // field@0020
000b8e: 6e20 1400 0200                           |0013: invoke-virtual {v2, v0},
Lcom/anssi/secret/RC4;.crypt:([B)[B // method@0014
```

Le chiffrement est donc en RC4, cependant, en observant le code de l'implémentation RC4, on constate qu'il s'agit d'un code RC4 classique exception faite de son initialisation qui, juste après avoir initialisé le tableau de permutation à l'aide de la clef de chiffrement/déchiffrement, va effectuer *3072* nouvelles permutations (sans utilisation de vecteur extérieur, exactement de la même façon que lors du chiffrement/déchiffrement RC4).
Ces permutations son implémentées dans la méthode getbyte. Ceci a le même effet que d'utiliser une graine constante pour initialiser l'algorithme, cela ne l'affaiblit pas. On constate également que cela a pour conséquence de laisser les constantes internes du RC4 servant d'index dans la table de permutation déjà initialisées et non pas null lors du début du chiffrement comme c'est l'usage.
Il est temps maintenant d'analyser la librairie libhello-jni.so à l'aide d'IDA et de gdb. Tout d'abord on constate que les noms de fonctions son relativement illisibles, "**`_8j3zIX`**" par exemple. On trouve rapidement une fonction baptisée "**`Java_com_anssi_secret_SecretJNI_deriverclef`**", voici la méthode native appelée par le code ***Dalvik***. Celle-ci n'étant appelée nulle part dans le code de la librairie, IDA ne peut deviner si il s'agit d'une partie de code, et si celle-ci est en *Thumb*(16) ou en *ARM*(32) (Nous y reviendrons plus tard). Il apparaît ici qu'il s'agit d'une fonction en *Thumb* 16Bits. Dans cette fonction, on observe un certain nombre d'appels à des sous fonctions internes aux noms improbables (exception faite des appels a strlen et sprintf) ainsi qu'un certain nombre d'appels à des fonctions externes. Celles-ci apparaissent sous la forme d'instruction "**`BLX registre`**". Le premier appel par exemple se présente de la façon suivante :

```
LDR     R2, [R0]
MOVS    R3, 0x2A4
LDR     R3, [R2,R3]
BLX     R3
```

Si l'on regarde le prototype d'une fonction JNI on a le premier argument valant "JNIEnv* env" puis le pointeur "self" caractéristique des langages objet "jobject thiz", vient ensuite les différents arguments. On conclu ici que la table de type JNIEnv contient toutes les adresses de tous les appels de fonctions externes misent à disposition d'une fonction Native JNI. Ainsi le premier appel de fonction externe de la fonction "**`Java_com_anssi_secret_SecretJNI_deriverclef`**" est la fonction d'index *0x2A4* dans la table JNIEnv.

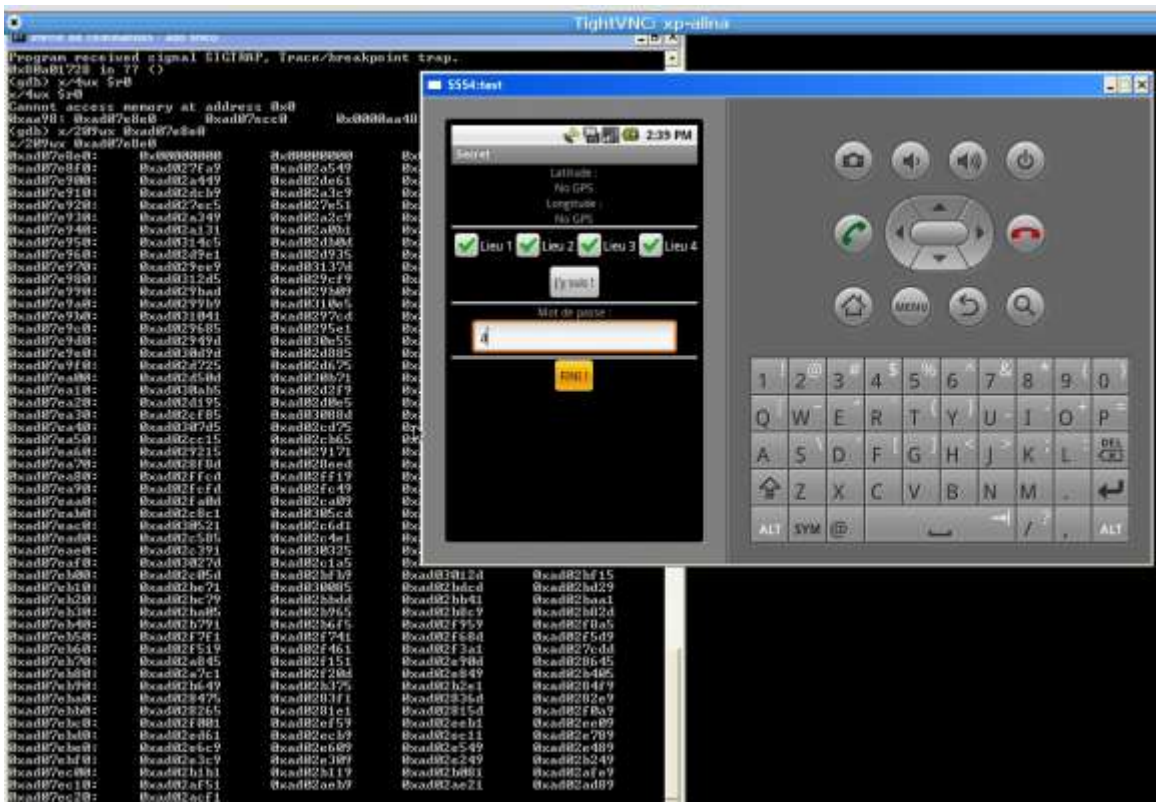
Il va nous falloir résoudre ces différents appels externes afin d'interpréter le fonctionnement de notre fonction. D'une manière générale, GDB nous aidera grandement pour observer le fonctionnement de notre fonction. Malheureusement le gdbserver n'a pas daigné fonctionner, impossible de débugger de façon distante, il aura donc fallu charger un GDB compilé en statique pour ***Android***.
On charge ce fichier par la commande "**`adb push gdb /data/`**". On peut alors utiliser la commande "**`adb shell`**" afin d'exécuter des commandes shell sur le périphérique ***Android*** émulé. On peut dès lors, comme sous un Linux, effectuer un "**`ps`**" pour obtenir le PID de notre processus cible (com.anssi.secret). Et en observer le mapping mémoire à l'aide du procfs :

```
cat /proc/<PID>/maps

80a00000-80a04000 r-xp 00000000 1f:01 669 /data/data/com.anssi.secret/lib/libhello-jni.so
80a04000-80a05000 rwxp 00003000 1f:01 669 /data/data/com.anssi.secret/lib/libhello-jni.so
```

Puis s'y attacher avec le GDB.

Il convient ici d'introduire un peut le fonctionnement de l'**ARM**, ce processeur peut traiter des instructions (opcode) d'une taille de 16Bits dites *Thumb* plus rapides mais au jeu d'instructions réduit et des instructions dites *ARM* encodées sur 32 Bits plus lentes.
Il existe également sur les versions récentes de l'**ARM** le *Thumb II* étendant le jeu d'instructions *Thumb*.
Tout binaire peut mixer ces jeux d'instructions à sa convenance. Le processeur intègre un flag *Thumb* dans son registre d'état (équivalent de l'eflag **Intel**) afin de définir le mode de fonctionnement courant. Le flag "*Thumb*", c'est son nom, est mis à 1 en mode *Thumb* et à 0 en mode *ARM*. Afin de basculer d'un mode à l'autre, un programme peut utiliser des opcode de branchement **`BX`** et **`BLX`** (le X pour eXchange) tout comme lors d'une restauration du Program Counter depuis la pile lors d'un retour de fonction, le mode sera susceptible de changer. Dans la mesure où les opcodes sont d'une taille de 2 octets (16Bits) ou de 4 octets (32Bits), les adresses des sauts dans du code sont donc des multiples de deux. Ainsi, le tout dernier bit de poids faible de ces adresses de code n'est pas utilisé. Celui-ci est du coup "recyclé" lors d'instructions pouvant changer de mode pour indiquer la nouvelle valeur du flag *Thumb* lors de l'appelle de fonction.
Ainsi une opcode **`BLX`** sautant à une adresse impaire, par exemple, indique que la fonction appelée est écrite en mode *Thumb*.
C'est également pour cette raison que si IDA ne voyant aucune fonction appeler la fonction "**`Java_com_anssi_secret_SecretJNI_deriverclef`**" ne peut conclure quant au mode à appliquer pour son désassemblage. (En réalité pas seulement car il existe de nombreuse méthode heuristiques relativement simple pour deviner le mode d'une fonction.)

Sous **ARM**, les anciennes versions de GDB sont clairement plus flexibles, à ce que j'en ai constaté, que le nouveau moteur de GDB. Celui-ci fait tout automatiquement mais est très difficile à forcer en cas d'erreur. Ces anciennes versions de GDB donc, adoptent la même nomenclature. Ainsi, désassembler une fonction par la commande "**`x/10i $pc`**" nous en fourniras le code en mode *ARM*, alors que la désassembler par la commande "**`x/10i $pc+1`**" nous en fournira la version en mode *Thumb*. Il en va de même lors de la pose de breakpoints logiciels, constitués d'une opcode dédiée, exigeant de connaître la taille de l'opcode courante et donc le mode en cours).



Donc, une fois attaché au processus avec GDB, on part de l'adresse de base de la librairie *0x80a00000* à la quelle on ajoute l'adresse de notre fonction, puis 1 car elle est en mode *Thumb*. Cela nous donne l'adresse du breakpoint à pauser dans GDB, on laisse alors le processus poursuivre son exécution par la commande "**`c`**".

```
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.

0xafe0da04 in __futex_wait () from /system/lib/libc.so
(gdb) b *0x80a01729
b *0x80a01729
Breakpoint 1 at 0x80a01729
(gdb) c
c
Continuing.
```

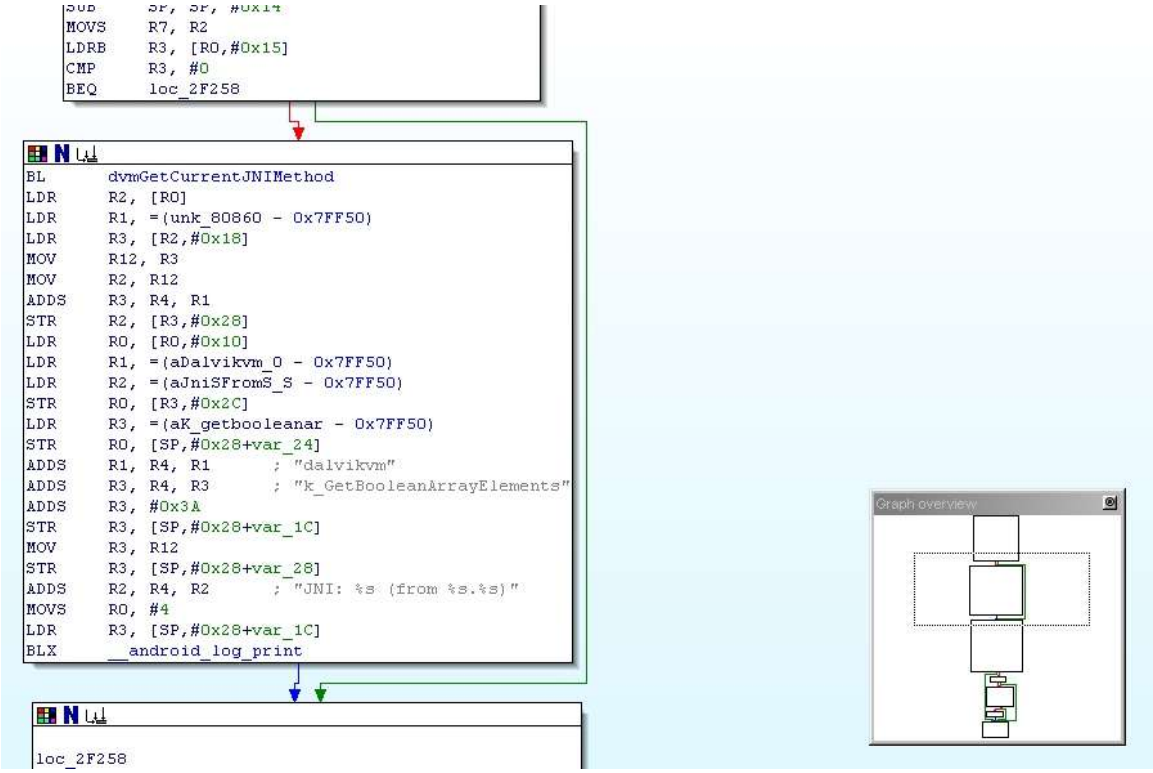
Il nous faut maintenant fournir à l'émulateur des coordonnées GPS, ceci en s'y connectant sur son port local, puis en passant des coordonné arbitraires par la commande de localisation "**`geo fix 1 1`**".
On peut alors valider quatre fois notre emplacement en validant le bouton "J'y suis !". Ceci fait on renseigne un mot de passe, arbitraire également, puis, il suffit de valider le bouton "FINI" afin de valider le tout. Une fois cela fait, le code ***Dalvik*** va appeler la méthode Native "**`Java_com_anssi_secret_SecretJNI_deriverclef`**" et GDB va alors s'arrêter à son début (sur notre breakpoint). A cet instant nous pouvons alors observer la table JNIEnv contenant l'ensemble des fonctions mises à disposition d'une méthode native JNI :

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x80a01728 in ?? ()
(gdb) x/4wx $r0
x/4wx $r0
Cannot access memory at address 0x0
0xaa98: 0xad07e8e0      0xad07ecc0      0x0000aa48      0x00000003
(gdb) x/209wx 0xad07e8e0
0xad07e8e0:     0x00000000      0x00000000      0x00000000      0x00000000
0xad07e8f0:     0xad027fa9      0xad02a549      0xad02887d      0xad02a4c9
0xad07e900:     0xad02a449      0xad02de61      0xad02dde1      0xad02dd49
0xad07e910:     0xad02dcb9      0xad02a3c9      0xad02dc1d      0xad027f35
0xad07e920:     0xad027ec5      0xad027e51      0xad0286cd      0xad027dd9
0xad07e930:     0xad02a349      0xad02a2c9      0xad02a249      0xad02a1c9
0xad07e940:     0xad02a131      0xad02a0b1      0xad027d61      0xad02db9d
0xad07e950:     0xad0314c5      0xad02db0d      0xad02da7d      0xad02a031
0xad07e960:     0xad02d9e1      0xad02d935      0xad031421      0xad029f8d
0xad07e970:     0xad029ee9      0xad03137d      0xad029e45      0xad029da1
0xad07e980:     0xad0312d5      0xad029cf9      0xad029c51      0xad031231
0xad07e990:     0xad029bad      0xad029b09      0xad031189      0xad029a61
0xad07e9a0:     0xad0299b9      0xad0310e5      0xad029915      0xad029871
0xad07e9b0:     0xad031041      0xad0297cd      0xad029729      0xad030f9d
0xad07e9c0:     0xad029685      0xad0295e1      0xad030ef9      0xad029541
0xad07e9d0:     0xad02949d      0xad030e55      0xad0293f9      0xad029355
0xad07e9e0:     0xad030d9d      0xad02d885      0xad02d7d5      0xad030ce5
0xad07e9f0:     0xad02d725      0xad02d675      0xad030c29      0xad02d5c1
0xad07ea00:     0xad02d50d      0xad030b71      0xad02d45d      0xad02d3ad
0xad07ea10:     0xad030ab5      0xad02d2f9      0xad02d245      0xad0309fd
0xad07ea20:     0xad02d195      0xad02d0e5      0xad030945      0xad02d035
0xad07ea30:     0xad02cf85      0xad03088d      0xad02ced5      0xad02ce25
0xad07ea40:     0xad0307d5      0xad02cd75      0xad02ccc5      0xad03071d
0xad07ea50:     0xad02cc15      0xad02cb65      0xad02cab9      0xad0292b5
0xad07ea60:     0xad029215      0xad029171      0xad0290d1      0xad02902d
0xad07ea70:     0xad028f8d      0xad028eed      0xad028e4d      0xad028dad
0xad07ea80:     0xad02ffcd      0xad02ff19      0xad02fe65      0xad02fdb1
0xad07ea90:     0xad02fcfd      0xad02fc49      0xad02fb89      0xad02facd
0xad07eaa0:     0xad02fa0d      0xad02ca09      0xad030675      0xad02c965
0xad07eab0:     0xad02c8c1      0xad0305cd      0xad02c81d      0xad02c779
0xad07eac0:     0xad030521      0xad02c6d1      0xad02c629      0xad030479
0xad07ead0:     0xad02c585      0xad02c4e1      0xad0303cd      0xad02c439
0xad07eae0:     0xad02c391      0xad030325      0xad02c2ed      0xad02c249
0xad07eaf0:     0xad03027d      0xad02c1a5      0xad02c101      0xad0301d5
0xad07eb00:     0xad02c05d      0xad02bfb9      0xad03012d      0xad02bf15
0xad07eb10:     0xad02be71      0xad030085      0xad02bdcd      0xad02bd29
0xad07eb20:     0xad02bc79      0xad02bbdd      0xad02bb41      0xad02baa1
0xad07eb30:     0xad02ba05      0xad02b965      0xad02b8c9      0xad02b82d
0xad07eb40:     0xad02b791      0xad02b6f5      0xad02f959      0xad02f8a5
0xad07eb50:     0xad02f7f1      0xad02f741      0xad02f68d      0xad02f5d9
0xad07eb60:     0xad02f519      0xad02f461      0xad02f3a1      0xad027cdd
0xad07eb70:     0xad02a845      0xad02f151      0xad02e90d      0xad028645
0xad07eb80:     0xad02a7c1      0xad02f20d      0xad02e849      0xad02b405
0xad07eb90:     0xad02b649      0xad02b375      0xad02b2e1      0xad0284f9
0xad07eba0:     0xad028475      0xad0283f1      0xad02836d      0xad0282e9
0xad07ebb0:     0xad028265      0xad0281e1      0xad02815d      0xad02f0a9
0xad07ebc0:     0xad02f001      0xad02ef59      0xad02eeb1      0xad02ee09
0xad07ebd0:     0xad02ed61      0xad02ecb9      0xad02ec11      0xad02e789
0xad07ebe0:     0xad02e6c9      0xad02e609      0xad02e549      0xad02e489
0xad07ebf0:     0xad02e3c9      0xad02e309      0xad02e249      0xad02b249
0xad07ec00:     0xad02b1b1      0xad02b119      0xad02b081      0xad02afe9
0xad07ec10:     0xad02af51      0xad02aeb9      0xad02ae21      0xad02ad89
0xad07ec20:     0xad02acf1
```

Nous dumpons les *209* premiers pointeurs de la table JNIEnv car l'index le plus éloigné utilisé par notre fonction est *0x340* (0x344 / 4 = 0xd1 = *209*).
On couvre ainsi tous les appels externes de la fonction. Notons que certains pointeurs n'utilise pas directement un index mais calcul celui-ci dynamiquement :

```
LDR     R2, [R7]
MOVS    R3, #0xC8
LSLS    R3, R3, #2              ; Index = 0xC8 * 2
LDR     R4, [R2,R3]
BLX     R4
```

Si on confronte les pointeurs précédemment obtenus avec la map du processus on constate qu'ils pointent tous dans la librairie libdvm.so; il s'agit de la librairie interprétant les opcodes ***Dalvik***, elle embarque l'intégralité de l'interpréteur ***DalvikVM***. Si on considère le premier appel externe effectué, son index est *0x2A4*, on fait donc appel à la fonction *0xad02f20d* (impaire, donc *Thumb*). En analysant la librairie libdvm.so avec IDA, on constate que cette fonction (comme les autres appelées), comporte une capacité de log par le biais de la fonction **`__android_log_print`** :

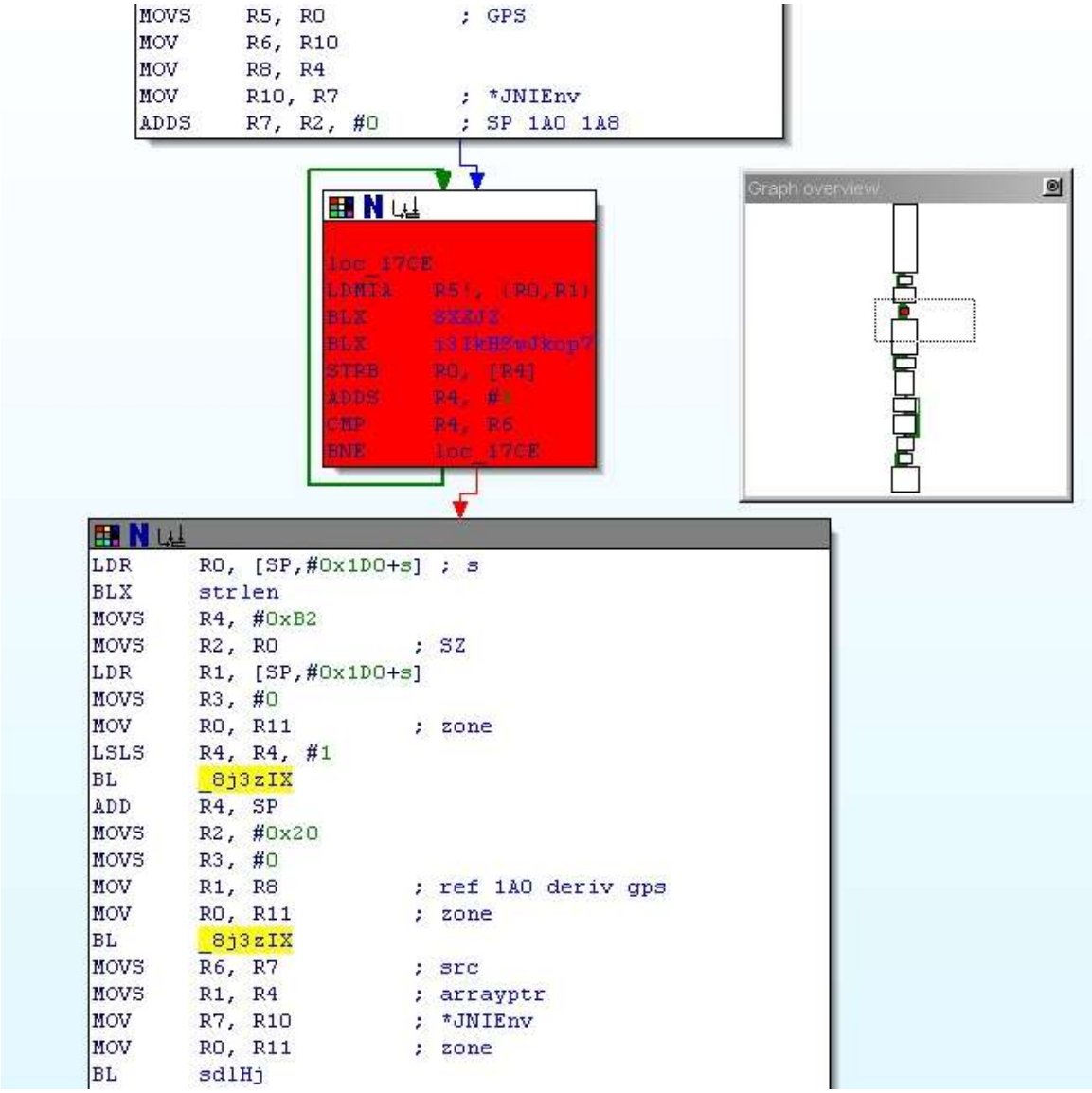


On voit ici que le paramètre de la chaîne de format "**`JNI: %s (from %s.%s)`**" est l'adresse de la chaîne de caractère "**`aK_getbooleanar`**" à laquelle on ajoute *0x3A* ("**`ADDS R3, #0x3A`**"). Si on observe ce qui se place à cette adresse on trouve la chaîne de caractère "**`GetStringUTFChars`**",0.

Le premier appel externe est donc la fonction **`GetStringUTFChars`**. Si on résout les appels suivant de la même manière on trouve pour notre fonction les appels externes suivants (dans l'ordre d'appel) :

```
GetStringUTFChars          -> obtention du mot de passe (r2 : premier argument)
NewByteArray
GetDoubleArrayElements     -> obtention de la table de coordonne GPS
                           -> (r3 : deuxième argument)
SetByteArrayRegion
FindClass
GetStaticMethodID
CallStaticVoidMethod
GetByteArrayRegion
ExceptionClear
ReleaseStringUTFChars
NewStringUTF
```

Avec ces nouvelles indications nous pouvons poursuivre l'analyse de notre fonction. En premier lieu, on récupère le mot de passe à l'aide de la fonction "**`GetStringUTFChars`**", on en stocke le pointeur en **`[SP+8]`**, ce pointeur n'est alors plus référencé qu'à un seul endroit où il sera utilisé :

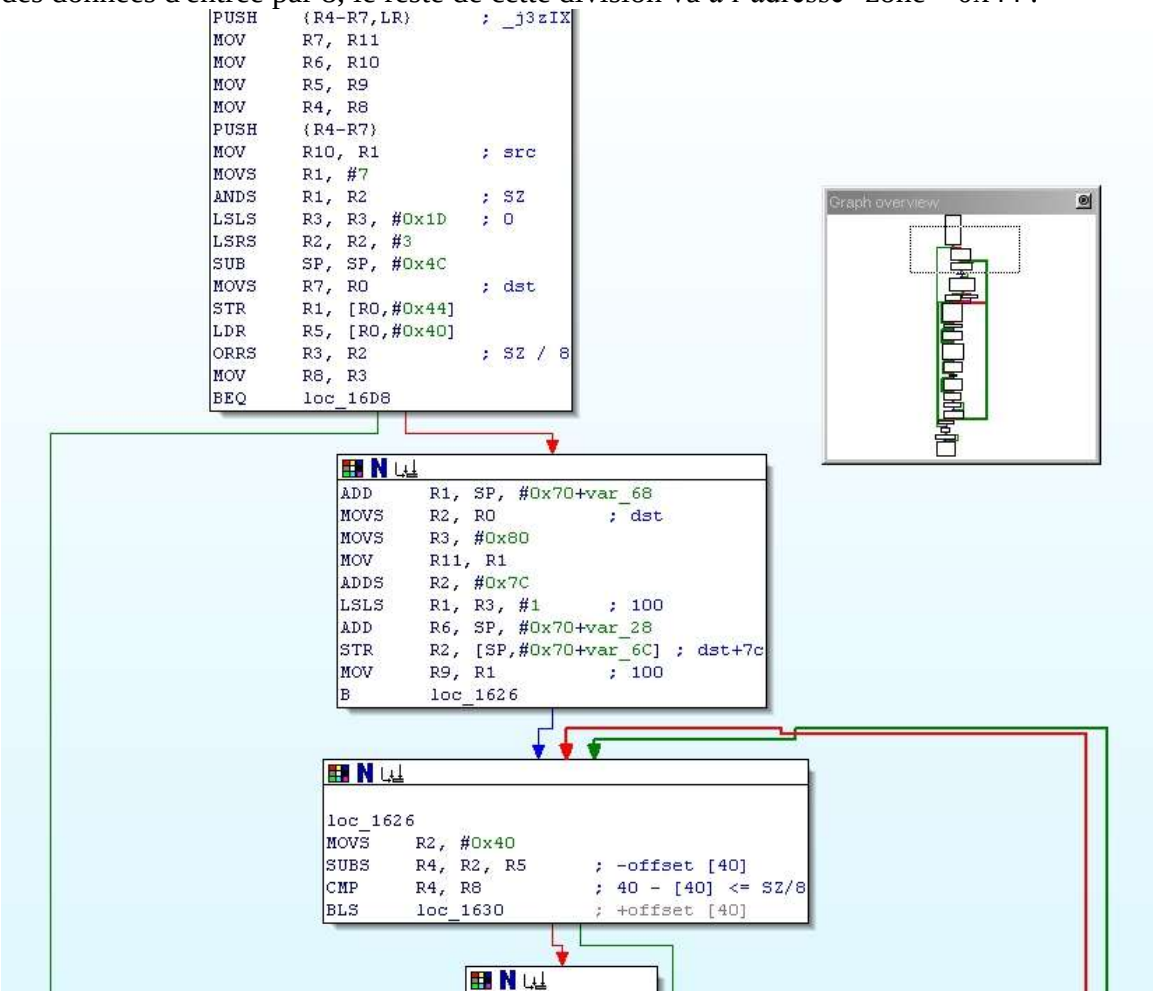


On extrait la taille de ce mot de passe à l'aide d'un **`strlen`**, puis on le donne en argument à la fonction interne "**`_8j3zIX`**". La fonction "**`_8j3zIX`**" semble prendre quatre arguments.

Le premier que l'on nomme "zone" dans l'IDA semble être une zone de donnée utilisée en lecture/écriture par la fonction, celle-ci, reflétant une partie de l'état interne de l'algorithme de dérivation de clef.
Le second semble être les données utilisées en entrée de l'algorithme (ici le mot de passe).
Le troisième est la taille de cette entrée.
Le quatrième est toujours NULL.

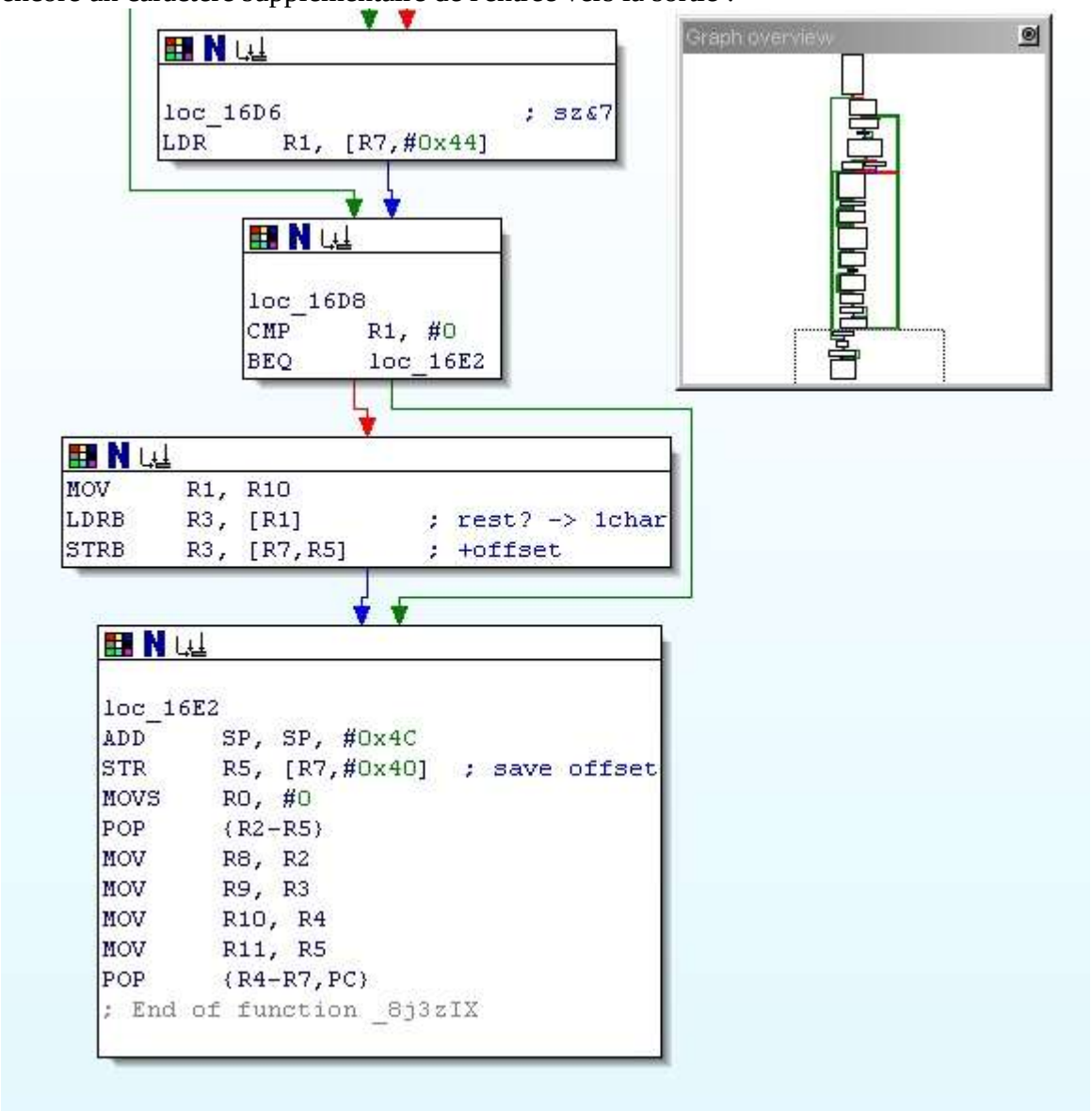
La fonction "**`_8j3zIX`**" n'est appelée que deux fois, une première fois avec le mot de passe et la seconde fois comme on le verra plus tard avec les données GPS modifiées.

Observons le fonctionnement de "**_8j3zIX**", cette fonction commence par diviser la taille des données d'entrée par 8, le reste de cette division va à l'adresse "zone"+0x44 :



Si les données d'entrée sont de taille supérieure à 7, on va effectuer une copie d'autant de caractères de l'entrée vers la sortie qu'il y a de bloc de 8 caractères dans l'entrée.
Pour cela l'algorithme utilise "zone"+0x40 comme un index mémorisant le nombre de caractères actuellement inscrit dans la sortie.
Cet index est initialisé à 0 lors de l'appel en amont d'une fonction d'initialisation de l'état interne de l'algorithme de dérivation de la clef. On ne l'analysera pas plus en détail, dans la mesure où elle s'effectue sans dépendre d'aucun argument, ni du mot de passe ni des coordonnées GPS.
On voit également, que si notre "zone" tampon de destination est remplie, (il fait une taille totale de *0x40* octets) on enchaîne sur d'autres mécanismes de dérivation plus complexes. Nous ne nous attarderons pas sur cette partie de l'algorithme, en effet, pour que cela se produise, il faut que l'on ai déjà placé à minima *0x40* octet dans la "zone" destination. Ceci implique donc, que notre clef ait une taille minimum de 0x40 *8 soit *512* octets. Taper un mot de *512* caractères sur un clavier de téléphone semble, en soit, un réel challenge… à moins d'être champion de SMS…
On va donc ignorer cette partie de l'algorithme.

Dans le cas contraire, si les données d'entrée sont de taille inférieure à 8, on ne place rien dans la "zone" de destination. Quoi qu'il en soit, la fonction "**`_8j3zIX`**" se finit alors en contrôlant le reste de la taille de l'entrés divisée par 8 ; si ce reste n'est pas null, on copie encore un caractère supplémentaire de l'entrée vers la sortie :



On constate que lors de cette opération, la fonction n'incrémente pas l'index en "zone"+0x40.

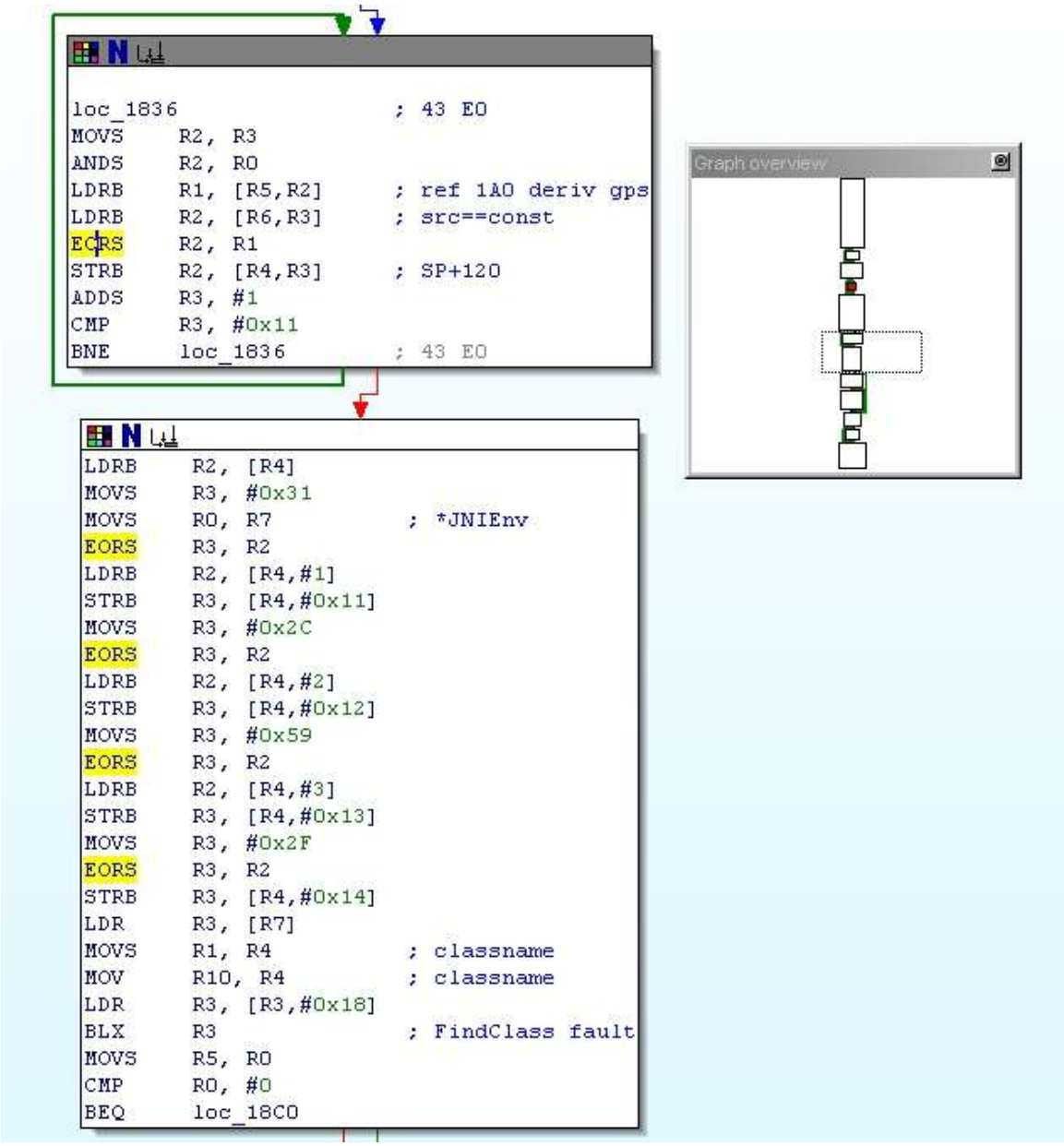
Hors ; le second appel à la fonction "**`_8j3zIX`**" est effectué juste après, et ce, avec une taille hardcodée de *0x20* (impliquant donc une copie de 4 caractères depuis les données présumées GPS). Dans la mesure où, précédemment, l'index en "zone"+0x40 n'a pas été incrémenté, l'éventuel rajout d'un caractère dans la "zone" de sortie si la taille de notre mot de passe n'est pas un multiple de 8, sera de toute façon écrasée par la suite et n'entre donc pas en ligne de compte.

Ainsi, si on considère une taille de clef inférieure à *512* octets, on sait que seuls n/8 caractères sont pris en compte pour une clef de taille n. Pour une clef allant de 16 à 23 caractères, seuls les deux premiers sont pris en compte !
Cela va grandement faciliter le brute-force.

Observons maintenant le traitement concernant le GPS. Cet argument est extrait à l'aide de la fonction **`GetDoubleArrayElements`**. Puis, ce tableau de double est traité dans la boucle (en rouge dans l'IDA) pour initialiser la zone allant de **`SP+1A0 a SP+1A8`**.

Il s'agit de cette zone qui est ensuite passé en argument d'entrée lors du second appel à la fonction "**`_8j3zIX`**" mais seuls ses 4 premier octets sont pris en compte (en effet la taille est une constante : *0x20* octets).

On peut voir juste un peu plus bas dans la fonction "**`Java_com_anssi_secret_SecretJNI_deriverclef`**" le traitement appliqué à ces données :



Ces 8 octets sont alors xorées (**`EOR`** en **ARM**) de façon cyclique avec la constante de 17 octets harcodée dans la librairie : "**`f4 94 7d 75 17 ee 04 fb fe d4 63 3f 15 f2 12 fc b8`**".

Après quoi, les quatre premiers octets obtenus sont xorées à nouveau par "**`31 2C 59 2F`**" pour obtenir au final quatre caractères supplémentaires constituant une chaîne de 21 caractères au total. Hors, cette chaîne est utilisée par la suite pour trouver une classe ***Dalvik*** à l'aide de la fonction "**`FindClass`**". De plus on peut voir, lors de l'appelle à la fonction "**`GetStaticMethodID`**", que les trois premiers octets de cette chaîne sont repris comme étant le nom de la méthode recherchée. On observe également que la signature de celle-ci est la constante : "**`([B[B)V`**". Cette méthode est au final appelée à l'aide de la fonction "**`CallStaticVoidMethod`**".

Si on combine les deux opérations xor précédemment dissociées dans le code, on obtient :

```
key 00 11 22 33 44 55 66 77 00 11 22 33 44 55 66 77 00 00 11 22 33
cst f4 94 7d 75 17 ee 04 fb fe d4 63 3f 15 f2 12 fc b8 C5 B8 24 5A
```

Dans la mesure où c'est une chaîne de caractères, elle se finit par un 0.
Ceci implique que l'octet (33) de la clef vaut 0x5A, ce qui à son tour implique les deux caractères suivant dans que la chaîne résultante :

```
key 00 11 22 33 44 55 66 77 00 11 22 33 44 55 66 77 00 00 11 22 33
cst f4 94 7d 75 17 ee 04 fb fe d4 63 3f 15 f2 12 fc b8 C5 B8 24 5A
out -- -- -- 2f -- -- -- -- -- -- -- 65 -- -- -- -- -- -- -- -- 00
```

Sachant qu'il s'agit de caractères imprimables il est possible d'utiliser cette contrainte autant de fois q'un octet de la clef est utilisé. Par exemple, pour le premier octet on a quatre contraintes :

```
var ^ f4, var ^ fe, var ^ b8, var ^ C5
```

Le résultat de chacune doit être imprimable.
On peut résoudre cela à l'aide d'un script perl (ici on exige un minimum de 3 lettres) :

```
#!/usr/bin/perl
#$file_cipher_data=$ARGV[0];

#open(H_C_DATA, $file_cipher_data) || die("Could not open cypher data");
#while (<H_C_DATA>){$cipher_data.=$_;}
#close(H_C_DATA);

for ($i=0; $i<256; $i++) {
$t=0;

$u=0xf4;
if ( ((($u^$i) <= 0x5a)&&(($u^$i) >= 0x41)) || ((($u^$i) <= 0x7a)&&(($u^$i) >=
0x61)) ) {$t=$t+1;}

$u=0xfe;
if ( ((($u^$i) <= 0x5a)&&(($u^$i) >= 0x41)) || ((($u^$i) <= 0x7a)&&(($u^$i) >=
0x61)) ) {$t=$t+1;}

$u=0xb8;
if ( ((($u^$i) <= 0x5a)&&(($u^$i) >= 0x41)) || ((($u^$i) <= 0x7a)&&(($u^$i) >=
0x61)) ) {$t=$t+1;}

$u=0xC5;
if ( ((($u^$i) <= 0x5a)&&(($u^$i) >= 0x41)) || ((($u^$i) <= 0x7a)&&(($u^$i) >=
0x61)) ) {$t=$t+1;}

if ($t >= 3) {print $i ." ". chr(0xf4^$i) ." ". chr(0xfe^$i) ." ".
chr(0xb8^$i) ." ". chr(0xC5^$i);print "\n";}
}
```

On obtient la clef testée et les caractères résultant :

```
132 p z < A
```

```
134 r x > C
135 s y ? B
140 x r 4 I
141 y s 5 H
142 z p 6 K
144 d n ( U
145 e o ) T
146 f l * W
147 g m + V
149 a k - P
150 b h . S
151 c i / R
156 h b $ Y
157 i c % X
159 k a ' Z
(...)
```

Cette manipulation peut être reproduite pour chaque caractère de la clef. D'un autre coté cela n'aura pas été nécessaire; en effet un simple
`strings challv2|grep -e ".../.......e........"`

Nous aura rapidement aguillés sur le fait que cela ne pouvait être que la chaîne "**`com/anssi/secret/RC4`**", et en effet, une fois la clef calculée pour les 8 premiers caractères :

```
clef_gps : 0x97 0xFB 0x10 0x5a 0x76 0x80 0x77 0x88
```

Les quatre derniers qui en découlent correspondent bien à notre chaîne, confirmant que nous avons eut la bonne intuition.

Cette clef est dérivée (par la boucle en rouge dans l'écran IDA précédent), après plusieurs observations "empirique" à l'aide de gdb, semblent correspondre aux coordonnées GPS que l'on peut fournir à l'émulateur par la commande :

```
geo fix long lat
geo fix -1.6 48
geo fix 28.7 5
geo fix 40.7 37.7
geo fix 43.2 38
```

Soit les coordonnés :

```
48 Nord 1.6 Ouest
5 Nord 28.7 Est
37.7 Nord 40.7 Est
38 Nord 43.2 Est
```

Qui, (sauf pour le premier), ne semblent pas précisément correspondre aux énigmes du mail déchiffré en partie 4. Nous connaissons donc les coordonnées GPS, celles-ci impliquent l'appelle de la méthode "**`com`**" de la classe "**`com/anssi/secret/RC4`**" qui est définie dans le **`dexdump`** comme suit :

```
0009fc:                                         |[0009fc] com.anssi.secret.RC4.com:([B[B)V
000a0c: 2200 1900                               |0000: new-instance v0,
Lcom/anssi/secret/RC4; // class@0019
000a10: 7020 1200 1000                          |0002: invoke-direct {v0, v1},
Lcom/anssi/secret/RC4;.<init>:([B)V // method@0012
000a16: 6e20 1500 2000                          |0005: invoke-virtual {v0, v2},
Lcom/anssi/secret/RC4;.cryptself:([B)V // method@0015
000a1c: 0e00                                    |0008: return-void
```

C'est donc un RC4 (modifié à l'initialisation comme nous l'avons vu précédemment) de la clef dérivée. (Celle-ci étant à la foi clef et donnée du RC4). Ainsi, l'application effectue le chaînage d'appel suivant :

Dalvik (saisie d'info) -> JNI (deriverclef) -> Dalvik (SELF RC4 clef) ->
-> JNI (result_to_hexdump) -> Dalvik (RC4 du programme avec la clef précédemment obtenue)

Il ne nous manque plus qu'un seul élément, le mot de passe. Nous allons simplement le brute-forcer.
Notons également que suite au deux appels à la fonction "**`_8j3zIX`**", nous avons une nouvelle fonction "**`sdlHj`**" qui va à son tours dériver la combinaison résultante des deux appels à "**`_8j3zIX`**" (et donc du mot de passe et des informations GPS), c'est cette combinaison dérivée qui seras alors passée à la méthode "**`com`**" de la classe "**`com/anssi/secret/RC4`**" lors du retours à la ***DalvikVM.*** C'est également cette combinaison dérivée que nous allons chercher à capturer en sortie du brute-force. Cela n'exige pas d'analyser plus amplement la fonction "**`sdlHj`**".

6 BRUTE-FORCE

Pour cela, on va écrire une nouvelle classe ***Dalvik***. Celle ci va générer les différents mots de passe possibles, puis instancier la librairie native afin de lui faire générer la clef dérivée.
Dans un second temps, nous ré implémentons le RC4 modifié ainsi que le selfRC4 de la clef dans un script python.
On peut alors déchiffrer avec toutes les clefs générées, les 16 premiers octets (pour des raisons de rapidité) du programme chiffré et y chercher la signature ELF.
C'est celle-ci qui nous indiquera un brute-force fructueux.

Le premier problème rencontré lors de cette implémentation fut que la ***DalvikVM*** tuait notre brute-force à cause du nombre d'instanciations trop grand (une fois par clef) du tableau de doubles GPS. La VM n'accepte pas plus de *1024* références à cette table. L'erreur peut être observée à l'aide de la commande "**`logcat`**" sous ***Android***. Pour des raisons de rapidité, il m'a semblé plus simple de supprimer l'utilisation du tableau GPS, et ce en patchant la librairie JNI afin d'utiliser des coordonnées GPS hardcodées.

Ceci a été appliqué pour un mot de passe de 8 à 15 caractères (un seul caractère effectif), mais sans résultat. Voyons ce que cela donne une fois adaptée aux 2 premiers caractères, celle-ci brute-forcera donc les mots de passe allant de 16 à 23 caractères.

Tout d'abord, on repart de l'exemple "**`hello-jni`**" fournit dans le SDK, on renomme en **`SecretJNI.java`** la classe d'origine, on adapte le path pour qu'il corresponde à la nouvelle classe "**`./src/com/anssi/secret/`**", et on change les différentes informations dans les paramètres du projet Eclipse.

On renomme également la fonction "**`Java_com_example_hellojni_HelloJni_stringFromJNI`**" par "**`Java_com_anssi_secret_SecretJNI_deriverclef`**" dans le code de la librairie JNI.

Le code est alors :

```
/*
 * Copyright (C) 2009 The Android Open Source Project
*/
#include <string.h>
#include <jni.h>

jstring
Java_com_anssi_secret_SecretJNI_deriverclef( JNIEnv* env,
                                                  jobject thiz, jstring pass,
jdoubleArray gps )
{
    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

Le code du brute forcer s'écrit lui dans le fichier de classe **`SecretJNI.java`** :

```
/*
 * Copyright (C) 2009 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package com.anssi.secret;

import android.app.Activity;
import android.widget.TextView;
import android.os.Bundle;
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class SecretJNI extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);


        try{
        FileOutputStream fOut = openFileOutput("keys",
                MODE_WORLD_READABLE | MODE_APPEND);
        OutputStreamWriter osw = new OutputStreamWriter(fOut);

        for (int a=32; a<127; a++) {
            for (int b=32; b<127; b++) {
            String a_ = Character.toString((char)a);
            String b_ = Character.toString((char)b);
            String pass =
(String)a_.concat((String)b_.concat("PAD1PAD2PAD3PAD4X"));    //2 chars de test
                osw.write(deriverclef(pass)); //, new double[0][0]
            }
        }
        osw.flush();
        osw.close();

        TextView  tv = new TextView(this);
        tv.setText( "d0ne" );
        setContentView(tv);

        }
catch (Exception e)
{
    String err = e.toString();
    System.out.println(err);
}
```

```
    }

    /* A native method that is implemented by the
     * 'hello-jni' native library, which is packaged
     * with this application.
     */
    public native String  deriverclef(String pass);//,double[][] gps);

    static {
        System.loadLibrary("hello-jni");
    }
}
```

Dès lors que ces codes compilent, on peut passer à la modification de la librairie JNI. On applique quatre différentes modifications :
Note: l'intégralité du code des patches a été compilée depuis un ***Iphone***...

La première consiste à supprimer l'appel à la fonction "**GetDoubleArrayElements**" puisque son argument est maintenant null.

On remplace donc :

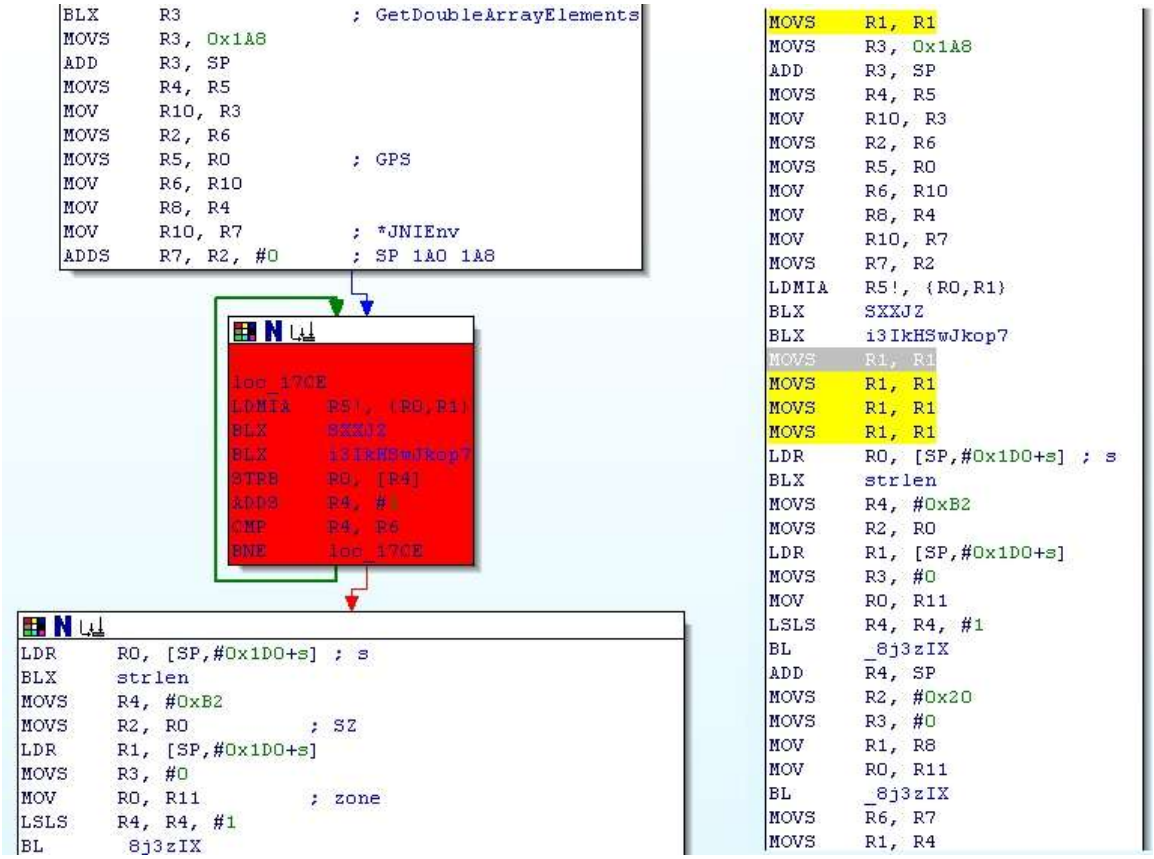
```
.text:000017B6 98 47                    BLX     R3; GetDoubleArrayElements
```

Par :

```
.text:000017B6 09 1C                    MOVS    R1, R1
```

(On utilise le "**MOVS R1, R1**" comme **NOP**).
La seconde consiste à supprimer la boucle traitant les données GPS puisqu'on va les fournir hardcodées ;

```
.text:000017D8 20 70                        STRB    R0, [R4]
.text:000017DA 01 34                        ADDS    R4, #1
.text:000017DC B4 42                        CMP     R4, R6
.text:000017DE F6 D1                        BNE     loc_17CE

.text:000017D8 09 1C                        MOVS    R1, R1
.text:000017DA 09 1C                        MOVS    R1, R1
.text:000017DC 09 1C                        MOVS    R1, R1
.text:000017DE 09 1C                        MOVS    R1, R1
```



On écrase alors le début de la fonction "**`i3IkHSwJkop7`**" afin qu'elle renseigne les données GPS prétraitées de façon hardcodées, on note que celle-ci est codée en *ARM* donc 32Bits cette fois-ci.
Le code original :

```
.text:00002230 81 20 A0 E1                 MOV     R2, R1,LSL#1
.text:00002234 02 26 92 E2                 ADDS    R2, R2, #0x200000
.text:00002238 0C 00 00 2A                 BCS     loc_2270
.text:0000223C 09 00 00 5A                 BPL     loc_2268
.text:00002240 3E 3E E0 E3                 MOV     R3, 0xFFFFFC1F
.text:00002244 C2 2A 53 E0                 SUBS    R2, R3, R2,ASR#21
.text:00002248 0A 00 00 9A                 BLS     loc_2278
```

Est écrasé par :

```
.text:00002230 0C 00 9F E5                 LDR     R0, =0x5A10FB97
.text:00002234 00 00 84 E5                 STR     R0, [R4]
.text:00002238 08 00 9F E5                 LDR     R0, =0x88778076
.text:0000223C 04 00 84 E5                 STR     R0, [R4,#4]
.text:00002240 1E FF 2F E1                 BX      LR
.text:00002240                 ; End of function i3IkHSwJkop7
.text:00002240
.text:00002240                 ; -------------------------------------------------
-------------------------
.text:00002244 97 FB 10 5A dword_2244      DCD 0x5A10FB97          ; DATA XREF:
i3IkHSwJkop7 r
```

```
.text:00002248 76 80 77 88 dword_2248      DCD 0x88778076          ; DATA XREF:
i3IkHSwJkop7+8 r
```

Pour finir, on supprime la partie traitant les xors et l'appelle de la méthode "**com**" (appelles au **FindClass** etc ..) car nous n'avons pas besoin d'effectuer le selfRC4, on ne l'implémente plus dans notre classe mais on le fait directement dans le code python.

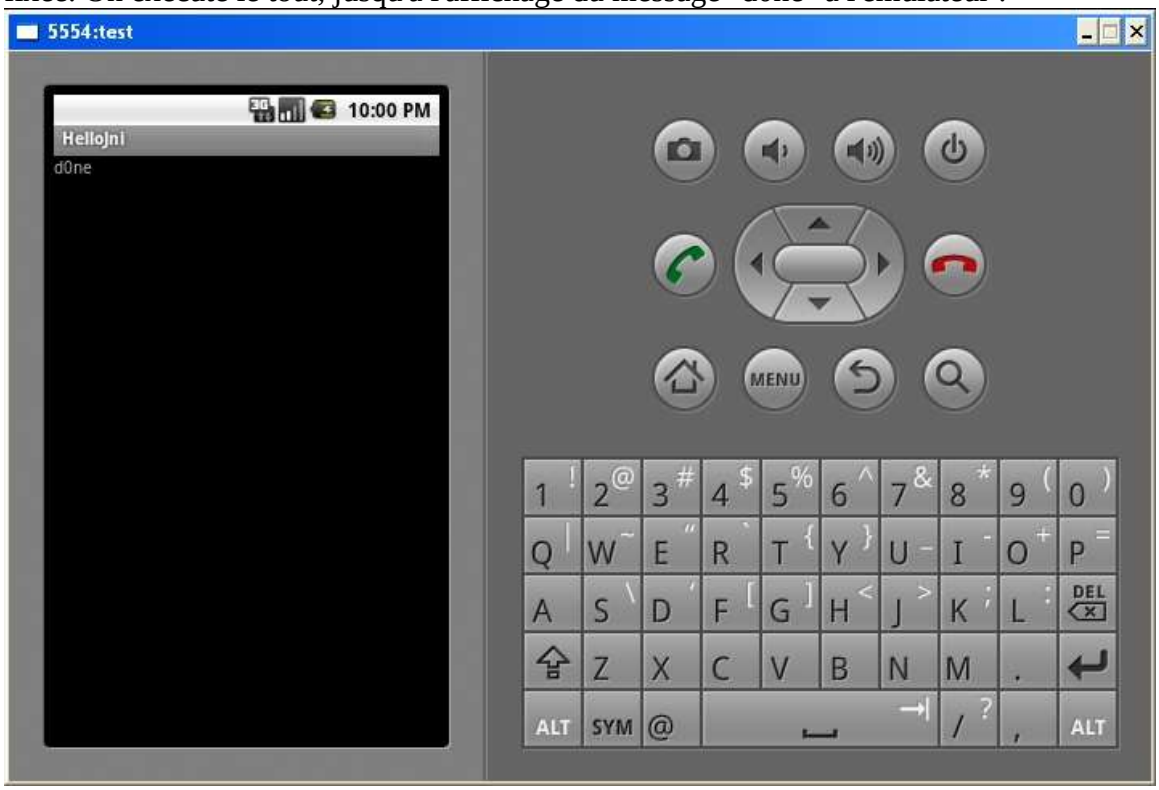
On remplace :

```
.text:00001836 1A 1C                    MOVS    R2, R3
```

Par :

```
.text:00001836 43 E0                    B       loc_18C0
```

Ceci fait on reconstruit notre paquet APK avec notre nouvelle librairie JNI fraîchement fixée. On exécute le tout, jusqu'a l'affichage du message "d0ne" à l'émulateur :



On peut alors extraire le fichier résultant afin de le traiter, avec la commande : "**adb pull /data/data/com.anssi.secret/files/keys c:\out**"
Il ne reste plus qu'à écrire la partie python, c'est simplement l'implémentation du RC4 donnée en exemple dans *wikipedia*, à la quelle on ajoute les *3072* permutations supplémentaires ainsi que le selfRC4 de la clef :

```
# -*- coding: iso-8859-1 -*-
class WikipediaARC4:
    def __init__(self, key = None):
        self.state = range(256) # initialisation de la table de permutation
        self.x = self.y = 0 # les index x et y, au lieu de i et j
```

```
        if key is not None:
            self.init(key)

    # Key schedule
    def init(self, key):
        for i in range(256):
            self.x = (ord(key[i % len(key)]) + self.state[i] + self.x) & 0xFF
            self.state[i], self.state[self.x] = self.state[self.x],
self.state[i]
        self.x = self.y = 0
        for i in xrange(3072):
            self.x = (self.x + 1) & 0xFF
            self.y = (self.state[self.x] + self.y) & 0xFF
            self.state[self.x], self.state[self.y] = self.state[self.y],
self.state[self.x]

    # Générateur
    def crypt(self, input):
        output = [None]*len(input)
        for i in xrange(len(input)):
            self.x = (self.x + 1) & 0xFF
            self.y = (self.state[self.x] + self.y) & 0xFF
            self.state[self.x], self.state[self.y] = self.state[self.y],
self.state[self.x]
            output[i] = chr((      ord(input[i]) ^
self.state[(self.state[self.x] + self.state[self.y]) & 0xFF]     ))
        return ''.join(output)

if __name__ == '__main__':
    test_vectors = [
                    [
"\xfe\x9a\xf0\x55\x34\x9d\x71\xb3"
"\x83\x8d\xce\x86\xf2\x65\x78\x85"
"\xcb\x57\x40\xce\x4b\x5e\x5a\x7a"
"\x88\xe9\x0c\x82\xd5\x3a\x7d\xc2"
, ' '], \

['\x47\x76\x89\xb3\xcb\x25\xeb\xa2\xb9\xd6\x71\xcb\x4a\x25\x6c\x07', ' '], \
]
    print
WikipediaARC4(WikipediaARC4(test_vectors[0][0]).crypt(test_vectors[0][0]).encod
e('hex')).crypt(test_vectors[1][0]).encode('hex').upper()
```

On peut maintenant scripter afin de trouver la clef valide.
Tout d'abord on extrait toutes les clefs du fichier keys à l'aide de la commande "**`split -b 64 -a4 ./keys`**".

On découpe alors notre script python de la façon suivante :

```
cat 1
# -*- coding: iso-8859-1 -*-
class WikipediaARC4:
    def __init__(self, key = None):
        self.state = range(256) # initialisation de la table de permutation
        self.x = self.y = 0 # les index x et y, au lieu de i et j

        if key is not None:
            self.init(key)

    # Key schedule
    def init(self, key):
```

```
        for i in range(256):
            self.x = (ord(key[i % len(key)]) + self.state[i] + self.x) & 0xFF
            self.state[i], self.state[self.x] = self.state[self.x], self.state[i]
        self.x = self.y = 0
        for i in xrange(3072):
            self.x = (self.x + 1) & 0xFF
            self.y = (self.state[self.x] + self.y) & 0xFF
            self.state[self.x], self.state[self.y] = self.state[self.y],
self.state[self.x]

    # Générateur
    def crypt(self, input):
        output = [None]*len(input)
        for i in xrange(len(input)):
            self.x = (self.x + 1) & 0xFF
            self.y = (self.state[self.x] + self.y) & 0xFF
            self.state[self.x], self.state[self.y] = self.state[self.y],
self.state[self.x]
            output[i] = chr((      ord(input[i]) ^ self.state[(self.state[self.x] +
self.state[self.y]) & 0xFF]     ))
        return ''.join(output)

if __name__ == '__main__':
    test_vectors = [
                    ["\
```

Puis :

```
cat 3
", ' '], \
                    ['\x47\x76\x89\xb3\xcb\x25\xeb\xa2\xb9\xd6\x71\xcb\x4a\x25\x6c\x07',
' '], \
]
    print
WikipediaARC4(WikipediaARC4(test_vectors[0][0]).crypt(test_vectors[0][0]).encode('hex')).
crypt(test_vectors[1][0]).encode('hex').upper()
```

Il nous suffit alors de lancer la boucle shell :

```
for i in $(ls x*); do cat ./1>./o.py; cat ./$i|sed -e
's/../\\x&/g'>>./o.py; cat ./3>>./o.py;echo $i;python o.py|grep -i
454c46; done;
```

Pour obtenir :

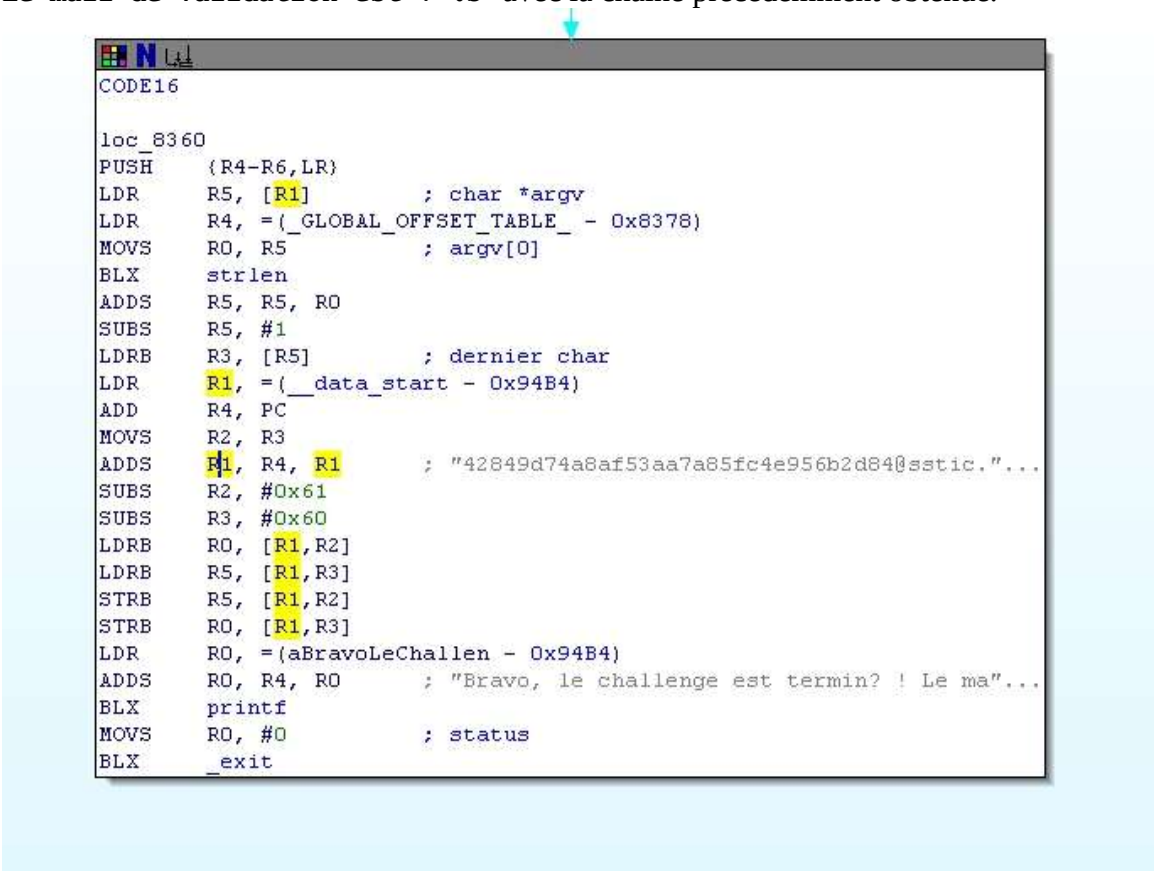
```
(...)
xagqp
xagqq
7F454C4601010100000000000000000
xagqr
(...)
```

La clef tant recherchée est donc :

```
cat xagqq
8d14b505adfc111e6cf420b19088b01d93321e93e31f7893cb4ad401560315d9
```

Correspondant au caractère du mot de passe : "**o7**"

On peut maintenant extraire le programme chiffré, en appliquant le même script python que précédemment pour trouver un petit binaire ELF compilé **ARM**. En le passant à l'IDA on constate que celui-ci se contente d'extraire le dernier caractère de son propre nom. Puis en soustrayant à celui-ci *0x60*, il s'en sert comme index pour intervertir deux caractères dans la chaîne : **`42849d74a8af53aa7a85fc4e956b2d84@sstic.org`**
Il conclue alors en nous affichant le message : "**`Bravo, le challenge est termin? ! Le mail de validation est : %s`**" avec la chaîne précédemment obtenue.



```
CODE16

loc_8360
PUSH    {R4-R6,LR}
LDR     R5, [R1]          ; char *argv
LDR     R4, =(_GLOBAL_OFFSET_TABLE_ - 0x8378)
MOVS    R0, R5            ; argv[0]
BLX     strlen
ADDS    R5, R5, R0
SUBS    R5, #1
LDRB    R3, [R5]          ; dernier char
LDR     R1, =(__data_start - 0x94B4)
ADD     R4, PC
MOVS    R2, R3
ADDS    R1, R4, R1        ; "42849d74a8af53aa7a85fc4e956b2d84@sstic."...
SUBS    R2, #0x61
SUBS    R3, #0x60
LDRB    R0, [R1,R2]
LDRB    R5, [R1,R3]
STRB    R5, [R1,R2]
STRB    R0, [R1,R3]
LDR     R0, =(aBravoLeChallen - 0x94B4)
ADDS    R0, R4, R0        ; "Bravo, le challenge est termin? ! Le ma"...
BLX     printf
MOVS    R0, #0            ; status
BLX     _exit
```

Il suffit alors de générer toutes les permutations possibles, l'une d'elle étant la bonne :

```
24849d74a8af53aa7a85fc4e956b2d84@sstic.org;
48249d74a8af53aa7a85fc4e956b2d84@sstic.org;
42489d74a8af53aa7a85fc4e956b2d84@sstic.org;
42894d74a8af53aa7a85fc4e956b2d84@sstic.org;
4284d974a8af53aa7a85fc4e956b2d84@sstic.org;
428497d4a8af53aa7a85fc4e956b2d84@sstic.org;
42849d47a8af53aa7a85fc4e956b2d84@sstic.org;
42849d7a48af53aa7a85fc4e956b2d84@sstic.org;
42849d748aaf53aa7a85fc4e956b2d84@sstic.org;
42849d74aa8f53aa7a85fc4e956b2d84@sstic.org;
42849d74a8fa53aa7a85fc4e956b2d84@sstic.org;
42849d74a8a5f3aa7a85fc4e956b2d84@sstic.org;
42849d74a8af35aa7a85fc4e956b2d84@sstic.org;
42849d74a8af5a3a7a85fc4e956b2d84@sstic.org;
42849d74a8af53aa7a85fc4e956b2d84@sstic.org;
42849d74a8af53a7aa85fc4e956b2d84@sstic.org;
```

```
42849d74a8af53aaa785fc4e956b2d84@sstic.org;
42849d74a8af53aa78a5fc4e956b2d84@sstic.org;
42849d74a8af53aa7a58fc4e956b2d84@sstic.org;
42849d74a8af53aa7a8f5c4e956b2d84@sstic.org;
42849d74a8af53aa7a85cf4e956b2d84@sstic.org;
42849d74a8af53aa7a85f4ce956b2d84@sstic.org;
42849d74a8af53aa7a85fce4956b2d84@sstic.org;
42849d74a8af53aa7a85fc49e56b2d84@sstic.org;
42849d74a8af53aa7a85fc4e596b2d84@sstic.org;
42849d74a8af53aa7a85fc4e965b2d84@sstic.org;
42849d74a8af53aa7a85fc4e95b62d84@sstic.org;
42849d74a8af53aa7a85fc4e9562bd84@sstic.org;
42849d74a8af53aa7a85fc4e956bd284@sstic.org;
42849d74a8af53aa7a85fc4e956b28d4@sstic.org;
42849d74a8af53aa7a85fc4e956b2d48@sstic.org
```

Il n'y a plus alors qu'a envoyer le mail à toutes ces adresses (encore un brute-force fort peu élégant..) pour finir la validation. En épluchant les logs du relais de messagerie, on trouve :

```
May  1 14:23:23 mail postfix/smtp[7244]: BAC9A1E3211:
to=<4284d974a8af53aa7a85fc4e956b2d84@sstic.org>,
relay=melen.sstic.org[88.191.97.98], delay=3, status=sent (250 OK
id=1O6Bj2-0004Tv-Ab)
```

Il s'avère donc que le mail final était `4284d974a8af53aa7a85fc4e956b2d84@sstic.org`, on en conclu que la dernière lettre du nom du binaire devait être un `'e'`.

Conclusion

Pour une toute première approche d'***Android***, c'était *touffu* mais bien sympas :)
Joli challenge !

Refs:

Structures du format ZIP
https://users.cs.jmu.edu/buchhofp/forensics/formats/pkzip.html
GDB compillé statique pour Android
http://sites.google.com/site/ortegaalfredo/android