Challenge SSTIC 2010: éléments de réponse

Arnaud Ebalard <arno@natisbad.org>

17 mai 2010

Résumé

Le challenge SSTIC 2010 consiste à analyser la copie intégrale de la mémoire physique d'un mobile tournant sous Android. L'objectif est d'y retrouver une adresse email en `@sstic.org`.

Après une étape initiale de reconstruction de deux applications Android (APK) fragmentées dans la copie mémoire, une première partie de crypto "ludique" permet d'accéder à des instructions. Ensuite, deux angles d'attaque s'offrent alors au participant.

Le premier consite à trouver les réponses à quatre questions (dont deux complexes) pour obtenir quatre jeux de coordonnées GPS et résoudre une épreuve de cryptographie "sérieuse" pour déchiffrer un message GPG contenant un mot de passe.

Le second consiste à réaliser le reverse d'une bibliothèque (ARM) et d'un fichier DEX Android de manière à retrouver les coordonnées GPS et le mot de passe évoqués précédemment.

Les détails techniques associés aux deux angles d'attaques sont présentés dans le document.

Table des matières

1 Introduction

Comme indiqué sur la page du challenge, le défi SSTIC 2010 consiste à analyser la copie intégrale de la mémoire physique d'un mobile tournant sous Android. L'objectif est d'y retrouver une adresse email en `@sstic.org`.

Ce document donne le détails des pistes suivies et des résultats obtenus dans le but d'obtenir cette adresse.

Sauf mention explicite du contraire dans le document, l'ensemble des étapes décrites a été réalisé sur un système Debian GNU/Linux (x86 32-bit).

A ce document sont attachés différents fichiers. En fonction de votre lecteur de PDF, voici comment y accéder :

- Si vous utilisez Acrobat Reader, vous pouvez accéder à ces fichiers depuis les menus : "Affichage" > "Panneaux de Navigation" > "Pièces jointes".
- Si vous utilisez Evince, vous pouvez sélectionner "Pièces jointes" en haut du panneau latéral sur la droite du document au lieu de "Index". Le panneau latéral peut-être affiché via le menu "Affichage"
- Autrement, vous pouvez utiliser `pdftk ce_document.pdf unpack_files` pour extraire tous les fichiers contenus dans le document dans le répertoire courant.

2 Analyse initiale de la mémoire

La première étape consiste à récupérer une copie de la mémoire :

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

$ sha256sum concours_sstic_2010
78c9ab57cdb8ba1eb7fde9a1d165fe86a6789320b63fb079f6ad8cd8dbebe037 \
concours_sstic_2010

$ ls -lh concours_sstic_2010
-rw-r--r-- 1 arno arno 23M Apr 21 19:13 concours_sstic_2010

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

$ 7z l -slt concours_sstic_2010 | tail -10
Path = challv2
Size = 100663296
Packed Size = 23667322
Modified = 2010-04-09 16:40:40
Attributes = ....A
CRC = 2B76B1BE
Encrypted = -
Method = LZMA:25
Block = 0
```

L'archive contient un fichier challv2 d'une centaine de Mo (compressé en LZMA). On l'extrait :

```
$ 7z e concours_sstic_2010
$ file challv2
challv2: data
```

Une analyse rapide des chaines de caractères contenues dans ces données (via **strings**) confirme qu'il s'agit bien d'un système Android :

```
$ strings challv2 | grep -c android
23096
$ strings challv2 | grep Linux
Linux version 2.6.29-00255-g7ca5167 (digit@digit.mtv.corp.google.com) \
(gcc version 4.4.0 (GCC) ) #9 Tue Dec 1 16:12:35 PST 2009
...
$ strings -e l challv2
```

Bien entendu, même si les données contiennent un grand nombre d'adresse email (et de caractères `@` en général), celle en `@sstic.org` n'est pas directement disponible dans la copie mémoire :

```
$ strings -e l challv2 | grep sstic.org
$ strings -e b challv2 | grep sstic.org
$ strings challv2 | grep sstic.org
```

L'option `-e` de **strings** permet d'extraire les chaines unicode de la mémoire. Une recherche sur les mots clés associés à SSTIC et à l'ANSSI apporte de nouvelles informations :

```
$ strings -e l challv2 | grep SSTIC
Challenge SSTIC0
Challenge SSTIC0
Challenge SSTIC0
Challenge SSTIC0
Challenge SSTIC0
Challenge SSTIC0
Challenge SSTIC
...
```

Un éditeur hexa (**hte** par exemple) permet de se rendre compte que ces chaines se trouvent en fait dans des certificats X.509 présents en mémoire.

La recherche de mots clés ('anssi' notamment) sur les chaines présentes en mémoire finit par faire apparaitre de nombreuses références à 2 applications Android (APK), vraisemblablement développées par l'ANSSI pour le challenge :

```
$ strings -e l challv2 | grep anssi
New package installed in /data/app/com.anssi.secret.apk
....
/data/app/com.anssi.secret.apk
```

```
...
/data/app/com.anssi.textviewer.apk
...
Starting com.anssi.textviewer
....
Starting com.anssi.secret
....
/data/data/com.anssi.secret/lib/libhello-jni.so
...

$ strings -e l challv2 | grep -c com.anssi.textviewer
137
$ strings -e l challv2 | grep -c com.anssi.secret
128
```

Une lecture exhaustive des chaines retournées par strings (plus de 600000) apparait comme une tâche fastidieuse même si elle est (a posteriori) susceptible de fournir des informations intéressantes.

La piste des 2 applications (**com.anssi.secret** et **com.anssi.textviewer**) vraisemblablement développées pour le challenge mérite d'être suivie dans un premier temps.

3 Reconstruction des applications (.apk)

3.1 Introduction rapide à Android

Le système d'exploitation Android de Google offre une architecture un peu particulière comparée aux OS et distributions plus classiques également basés sur le noyau Linux. De bonnes introductions à l'architecture d'Android sont disponibles ici et ici.

Les points importants (pour le challenge) concernant le système sont les suivants :

- Android est basé sur le noyaux Linux. Celui-ci fournit notamment le support du matériel, la gestion de l'énergie, ... **et de la mémoire**.
- Des bibliothèques (SSL, libc, ...) en espace utilisateur font le lien avec les applications et offrent des services de plus haut niveau. Il est à noter que la libc du système (connue sous le nom de bionic) n'est pas la libc classique que l'on trouve sur les sytèmes Linux.
- Android vise (pour le moment) les plateformes mobiles tels que les téléphones portables et n'est disponible que pour des architectures animées par des **processeurs ARM**.
- Les applications Android n'ont pas directement accès au processeur de la machine. Elles tournent dans une machine virtuelle spécifique au système : Dalvik. Les applications sont développées en Java et compilées vers un bytecode compris par Dalvik (différent du bytecode Java). Les applications compilées sont **au format dex**.

- Les applications sont fournies **sous forme de paquets (Application Package, i.e. APK)**. Le format des APK est comparable sur de nombreux aspects à celui des JAR. **Il s'agit d'une archive au format ZIP** contenant un .dex, des fichiers de ressources, un fichier de manifeste, un certificat, ...

Le contenu d'un APK dézippé est donné ci-dessous, à titre d'exemple :

```
$ unzip example.apk
$ find .
./resources.arsc
./META-INF
./META-INF/CERT.SF
./META-INF/MANIFEST.MF
./META-INF/CERT.RSA
./AndroidManifest.xml
./res
./res/drawable-mdpi
./res/drawable-mdpi/icon.png
./res/layout
./res/layout/main.xml
./res/drawable-ldpi
./res/drawable-ldpi/icon.png
./res/drawable-hdpi
./res/drawable-hdpi/icon.png
./classes.dex
```

3.2 Angles d'attaque pour la reconstruction

Deux applications Android développées pour le challenge semblent donc avoir été installées sur le système dont la copie mémoire nous a été fournie.

Il semble que les paquets de ces applications aient été chargés en mémoire (probablement lors de l'installation) et y résident encore (au moins partiellement). Pour pouvoir continuer, nous procédons à la reconstruction de ces deux applications à partie de la copie mémoire.

Le principal problème associé à la reconstruction de fichiers à partir de la copie mémoire découle de la fragmentation de celle-ci. Elle résulte directement de la gestion réalisée par le noyau Linux : la mémoire physique est découpée par celui-ci en page de 4Ko ; il fournit aux applications une vision différente (mémoire virtuelle). Un fichier mappé linéairement en mémoire virtuelle se trouve éventuellement réparti sur plusieurs pages non contigues en mémoire physique.

Pour remonter aux deux applications Android qui nous intéressent, il existe au moins deux angles d'attaque possibles :

1. Retrouver les structures internes de gestion mémoire utilisées par le noyau Linux et reconstruire en les utilisant la mémoire virtuelle des différents

processus. En fournissant l'accès aux portions de mémoire virtuelle, cette approche permet d'obtenir une vue linéaire des fichiers mappés en mémoire indépendamment de leur type. Malgré tout, elle ne sera effective que pour les fichiers mappés en mémoire de processus lancés au moment de la création de la copie mémoire.

2. Considérer la copie mémoire comme un puzzle constitué de pièces de 4Ko et se baser sur les informations spécifiques au type de fichier cherché pour réassembler les morceaux. L'inconvénient principal de cette méthode est qu'elle requiert de comprendre la structure précise du ou des types de fichiers considérés. Son avantage, contrairement à la méthode précédente, est qu'elle peut permettre la reconstruction de fichiers présents dans des pages libres asssociés à d'anciens processus.

Un unique type de fichier étant à reconstruire (APK), le second angle d'attaque a été choisi.

Les APK étant en pratique de simples archives ZIP, la sous-section suivante décrit ce format de fichier. Ensuite sont décrits les détails d'implémentation et les problèmes rencontrés lors de la reconstruction.

3.3 Le format de fichier ZIP

Une bonne introduction au format de fichier ZIP est fournie par la page Wikipedia associée. Des détails supplémentaires sont disponibles ici.

Une archive ZIP est constituée des trois types d'éléments principaux suivants :

1. **ZIP End of Central Directory Record** : cet élément se situe à la fin de l'archive et donne des informations sur les éléments qui le précèdent directement (**ZIP Central Directory File Header**) et qui constituent avec lui le Central Directory (une sorte de table des matières de l'archive). Entre autres informations, le **ZIP End of Central Directory Record** donne la taille du Central Directory et le nombre de **ZIP Central Directory File Header** qu'il contient. Le **ZIP End of Central Directory Record** débute par 4 octets de signature : 0x06054b50
2. **ZIP Central Directory File Header** : plusieurs entrées de ce type précèdent le **ZIP End of Central Directory Record**. Chacune donne des détails sur un fichier contenu dans l'archive : CRC32 (de la version décompressée), taille du fichier compressé, taille du fichier décompressé, nom de fichier. Mais cette entrée fournit également la position du **ZIP Local File Header** dans l'archive (le fichier compressé précédé d'un header). Un **ZIP Central Directory File Header** débute par quatre octets de signature : 0x02014b50.
3. **ZIP Local File Header** : à chacun des fichiers de l'archive (ou répertoire) est associé une de ces entrées. Celle-ci fournit notamment des informations sur la méthode de compression utilisée, la date de modification, le CRC32 (de la version décompressée), la taille du fichier compressé, la taille du

fichier décompressé, le nom du fichier, ...Un **ZIP Local File Header** débute par quatre octets de signature : 0x04034b50.

L'image suivante [1] empruntée à l'article Wikipédia cité précédemment donne une vision graphique du format de fichier ZIP.

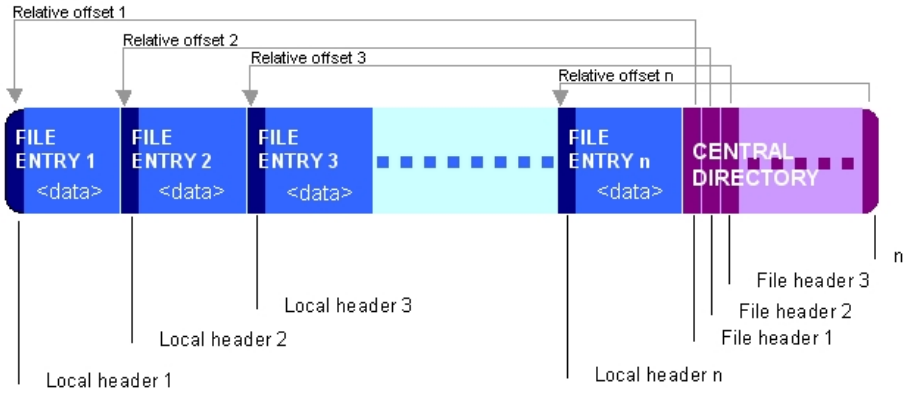

FIGURE 1 – Format de fichier ZIP

3.4 Détails techniques sur la récupération des APK

Comme évoqué précédemment et en utilisant les informations données sur le format de fichier ZIP, cette section offre quelques détails techniques sur la récupération des APK à partir de la copie mémoire. Un module Python (**zipfinder.py**) contenant le code utilisé est attaché au document.

La première étape consiste à parcourir la copie méoire à la recherche de la signature associée au **Zip End of Central Directory Header** (0x06054b50). Cet élément contient la taille totale du Central Directory, i.e. la taille de l'ensemble des **ZIP Central Directory Headers** qui le précèdent.

Si cette information place le premier Central Directory Header de cette archive ZIP dans la page où le **Zip End of Central Directory Header** a été trouvé, toutes les informations sur les fichiers de l'archive sont potentiellement disponibles. Si ce n'est pas le cas (indiquant que le Central Directory est réparti sur au moins 2 pages), nous ne tentons pas de reconstruire l'archive associée. Cette décision est motivée par le fait que les 2 APK qui nous intéressent ne font pas partie des fichiers dont le Central Directory est fragmenté.

Ensuite, chacun des ZIP Central Directory File Header des archives à reconstruire permet de créer une liste des fichiers contenus dans les archives à reconstruire. Pour chacun d'eux, leur nom, leur CRC32 et leurs tailles compressées et décompressées sont les information qui permettent de les discriminer.

1. disponible à http://en.wikipedia.org/wiki/File:ZIPformat.jpg, sous licence Creative Commons Attribution-ShareAlike 3.0

L'étape suivante consiste à parcourir à nouveau la mémoire, cette fois-ci à la recherche de la signature des **ZIP Local File Headers**. Si le CRC32, le nom de fichier, la taille du fichier compressé et la taille du fichier décompressé (toutes présentes dans cet élément) correspondent à ceux d'un fichier à récupérer, le travail de reconstruction suivant est effectué. Sinon, cet élément n'est pas considéré.

L'ensemble des éléments structurels du ZIP considéré jusqu'à présent n'ont que peu de chance de subir de la fragmentation, du fait de leur taille réduite. Les fichiers compressés à récupérer sont pour certains d'entre eux d'une taille supérieure à la taille d'une page. La fragmentation est donc garantie pour ceux-ci.

Malgré tout, pour les archives qui nous intéressent le plus, ces fichiers compressés restent de taille relativement faible (au plus 4 pages). Par exemple, la version compressée du fichier **lib/armeabi/libhello-jni.so** appartenant à l'un des deux APK intéressants a une taille de 9472 octets. Cet élément compressé est réparti sur quatre pages.

Pour un fichier compressé donné, l'idée est de tester toutes les combinaisons possibles du nombre de pages nécessaires. Si la décompression s'effectue correctement et que le CRC32 correspond à celui présent dans le **ZIP Local File Header**, cela signifie que la combinaison courante de page est la bonne.

La complexité de cette méthode est exponentielle en nombre de pages sur lesquelles le fichier compressé considéré est fragmenté. Le nombre total de pages dans la copie mémoire ($100663296/4096 = 24576$) rend cette méthode inutilisable sans optimisation particulière. Pour réduire la complexité, il suffit d'utiliser les deux faits suivants :

- *Les pages centrales des fichiers compressés contiennent par définition des données compressées, ce qui n'est pas le cas de la plupart des pages en mémoire.* Ceci permet de les discriminer assez facilement : une page qui se recompresse mal, i.e. dont la version compressée est plus longue que sa version initiale vaut la peine d'être considérée. Cette heuristique permet de créer un ensemble de 1271 pages intéressantes à utiliser comme pages centrales.
- *Un fichier compressé est suivi dans l'archive soit par un* **ZIP Local File Header** *(signature 0x04034b50), soit par un* **ZIP Central Directory File Header** *(signature 0x02014b50).* En cherchant la dernière page uniquement dans l'ensemble des pages contenant des signatures ZIP, ceci permet de passer de 24576 à 2071 pages.

Ces optimisations permettent par exemple de passer [2] de $24576^3 = 14843406974976$ à $1271^2 \cdot 2071 = 3345578311$ tests pour la reconstruction de la version compressée de **lib/armeabi/libhello-jni.so**.

La valeur précédente reste encore assez importante (chaque étape de test impose des opérations de concatenation et au moins une tentative de décompression).

2. La première page est celle contenant le ZIP Local File Header

Durant le développement du module **zipfinder.py** implémentant les heuristiques présentées ci-dessus, des tests fréquents ont imposés de réduire *temporairement* l'ensemble de pages chiffrées aux premières trouvées. En limitant cet ensemble au 50 premières, la reconstruction du fichier **libhello-jni.so** se déroule sans problème. Plus généralement, il se trouve que ces 50 premières pages [3] suffisent pour obtenir la reconstruction des 2 APK intéressants (associés aux applications TextViewer et Secret). En fait, cette valeur permet même la reconstruction de 7 APK complets, la procédure terminant en moins de 4 minutes sur une machine récente. Cette valeur de 50 a donc été conservée dans le module attaché à ce document.

```
>>> from zipfinder import *
>>> c, i = recover_zip_files("challv2")
[+] Looking for already encrypted pages
    => 50 pages in that pool                          # !! 50 premieres
[+] Looking for interesting pages (with zip headers)
    => 2071 pages in that pool
[+] Searching for ZIP Central Directory headers
    => 13 ZIP files to reconstruct
[+] Creating a list of interesting files
    => 98 files contained in those ZIP
[+] Starting processing to rebuild our local file headers
[ ] Recovering 4 blocks compressed file lib/armeabi/libhello-jni.so ...
[+] Recovered 4 blocks compressed file lib/armeabi/libhello-jni.so ...
[ ] Trying to recover 2 blocks compressed file res/layout/main.xml
[+] Recovered 2 blocks compressed file res/layout/main.xml ...
[ ] Trying to recover 3 blocks compressed file classes.dex
[-] Failed to recover 3 blocks compressed file classes.dex ...
Dropping classes.dex due to bad crc (len 5732)
Dropping resources.arsc due to bad crc (len 7632)
[ ] Trying to recover 2 blocks compressed file res/layout/fallback.xml
[+] Recovered 2 blocks compressed file res/layout/fallback.xml ...
[ ] Trying to recover 3 blocks compressed file classes.dex
[-] Failed to recover 3 blocks compressed file classes.dex ...
Dropping classes.dex due to bad crc (len 7651)
[ ] Trying to recover 3 blocks compressed file classes.dex
[+] Recovered 3 blocks compressed file classes.dex ...
[ ] Trying to recover 3 blocks compressed file classes.dex
[-] Failed to recover 3 blocks compressed file classes.dex ...
Dropping classes.dex due to bad crc (len 6019)
[ ] Trying to recover 2 blocks compressed file classes.dex
[+] Recovered 2 blocks compressed file classes.dex ...
[ ] Trying to recover 3 blocks compressed file classes.dex
[-] Failed to recover 3 blocks compressed file classes.dex ...
Dropping classes.dex due to bad crc (len 6283)
Dropping resources.arsc due to bad crc (len 7896)
[ ] Trying to recover 3 blocks compressed file classes.dex
[-] Failed to recover 3 blocks compressed file classes.dex ...
```

3. Il est certainement possible de diminuer encore cette valeur

```
Dropping classes.dex due to bad crc (len 7230)
[ ] Trying to recover 2 blocks compressed file classes.dex
[-] Failed to recover 2 blocks compressed file classes.dex ...
Dropping classes.dex due to bad crc (len 1504)
[ ] Trying to recover 3 blocks compressed file classes.dex
[-] Failed to recover 3 blocks compressed file classes.dex ...
Dropping classes.dex due to bad crc (len 9834)
Dropping res/layout/main.xml due to bad crc (len 899)
Dropping META-INF/CERT.SF due to bad crc (len 305)
Found 110 Local File headers in challv2
[+] Recovered 11 component of 11 in ZIP file
[-] Component classes.dex (len:6283 crc:3690722697) not recovered
[-] Component resources.arsc (len:7896 crc:685208511) not recovered
[+] Recovered 7 component of 7 in ZIP file
[+] Recovered 5 component of 5 in ZIP file
[+] Recovered 11 component of 11 in ZIP file
[+] Recovered 7 component of 7 in ZIP file
[-] Component classes.dex (len:7230 crc:390057870) not recovered
[+] Recovered 6 component of 6 in ZIP file
[-] Component classes.dex (len:6019 crc:2663298127) not recovered
[+] Recovered 13 component of 13 in ZIP file
[-] Component classes.dex (len:5732 crc:3749019707) not recovered
[-] Component resources.arsc (len:7632 crc:3524226205) not recovered
[-] Component classes.dex (len:7651 crc:2805418067) not recovered
[-] Component META-INF/ (len:2 crc:0) not recovered
[-] Component META-INF/MANIFEST.MF (len:71 crc:2310898753) not recovered
[-] Component classes.dex (len:9834 crc:3826679922) not recovered
[+] Final result: 7 complete, 6 incomplete
```

Les 2 listes retournées par recover_zip_files() contiennent respectivement des classes associées à des archives complètes et à des archives incomplètes (pour lesquelles un fichier au moins n'a pu être reconstruit). Il suffit ensuite d'exporter les APK sur disques :

```
>>> j = 0
>>> for f in c:
...     f.export('%02d.apk' % j)
...     j+=1
...
```

On se rend compte assez facilement en inspectant le contenu de la liste 'c' que **com.anssi.textviewer.apk** et **com.anssi.secret.apk** sont respectivement le premier et le dernier ZIP de la liste. Ceux-ci correspondent donc sur disque à 00.apk et 06.apk :

```
$ unzip 00.apk -d com.anssi.textviewer
Archive:  00.apk
  inflating: com.anssi.textviewer/res/layout/main.xml
  inflating: com.anssi.textviewer/res/raw/chiffre.txt
```

```
  inflating: com.anssi.textviewer/AndroidManifest.xml
 extracting: com.anssi.textviewer/resources.arsc
 extracting: com.anssi.textviewer/res/drawable-hdpi/icon.png
 extracting: com.anssi.textviewer/res/drawable-ldpi/icon.png
 extracting: com.anssi.textviewer/res/drawable-mdpi/icon.png
  inflating: com.anssi.textviewer/classes.dex
  inflating: com.anssi.textviewer/META-INF/MANIFEST.MF
  inflating: com.anssi.textviewer/META-INF/CERT.SF
  inflating: com.anssi.textviewer/META-INF/CERT.RSA

$ unzip 06.apk -d com.anssi.secret
Archive:  06.apk
  inflating: com.anssi.secret/classes.dex
  inflating: com.anssi.secret/AndroidManifest.xml
  inflating: com.anssi.secret/resources.arsc
   creating: com.anssi.secret/lib/
   creating: com.anssi.secret/lib/armeabi/
  inflating: com.anssi.secret/lib/armeabi/libhello-jni.so
   creating: com.anssi.secret/META-INF/
  inflating: com.anssi.secret/META-INF/MANIFEST.MF
  inflating: com.anssi.secret/META-INF/CERT.RSA
  inflating: com.anssi.secret/META-INF/CERT.SF
   creating: com.anssi.secret/res/
   creating: com.anssi.secret/res/layout/
  inflating: com.anssi.secret/res/layout/main.xml
```

Les deux archives obtenues sont attachées à ce document (ainsi que le module **zipfinder.py**).

3.5 Conclusion

La reconstruction des 2 APK du challenge a été réalisée en utilisant notamment les particularités du format de fichier ZIP et quelques heuristiques assez simples de manière à rendre possibles les étapes de brute force.

Cette méthode à fonctionné principalement du fait de la taille assez limitée des fichiers compressés contenus dans les deux archives à reconstruire. Si ceux-ci avaient été fragmentés sur un plus grand nombre de pages, il auraient peut-être été nécessaire de reconstruire en plus la mémoire des différents processus ou des trouver d'autres heuristiques.

4 Interaction avec les applications

4.1 Installation de l'émulateur Android

Le kit de développement (SDK) d'Android inclut un émulateur. Celui-ci permet d'installer et de tester des application Android. Le SDK est téléchargeable

ici.

Une fois le téléchargement effectué et l'archive décompressée, il suffit de lancer l'application graphique "Android SDK and AVD Manager" disponible dans le répertoire **tools**.

```
$ tar xzf android/android-sdk_r05-linux_86.tgz
$ cd android-sdk-linux_86
$ ./tools/android
```

Dans l'onglet "Available Packages", il suffit de sélectionner les éléments associés au SDK pour Android 2.1 comme présenté sur l'image ci-dessous puis demander l'installation (via "Install Selected").

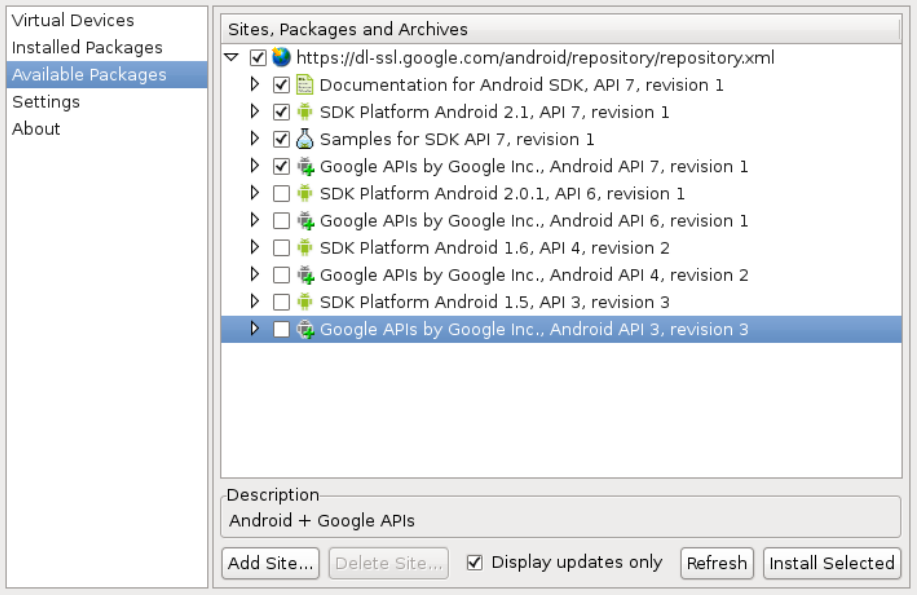

FIGURE 2 – Sélection des éléments associés au SDK Android 2.1 à installer

Il est à noter que si le message d'erreur suivant apparait, il suffit de positionner l'entrée `/proc/sys/net/ipv6/bindv6only` à `0`. Le lecteur intéressé par les détails est renvoyé au rapport de bug Debian #560044.

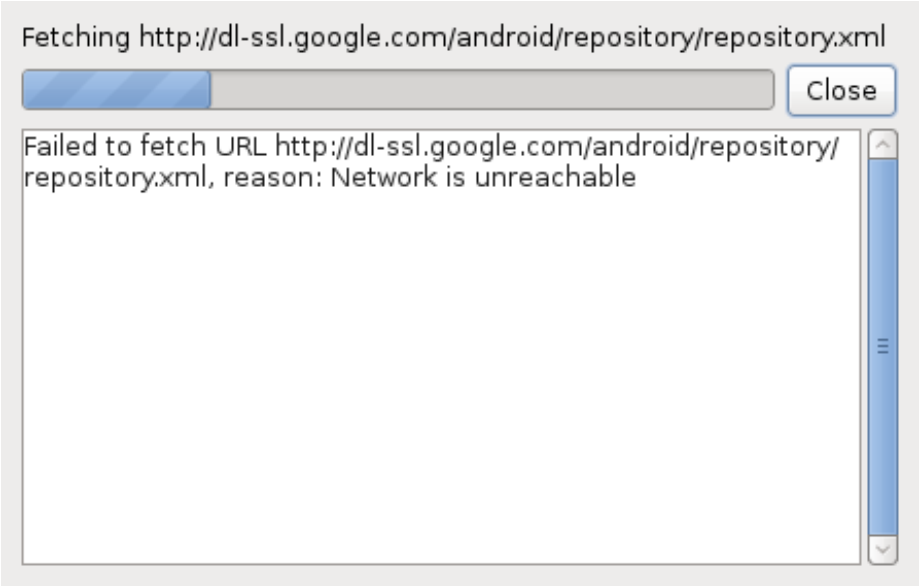

FIGURE 3 – Bug #560044

Une fois les éléments téléchargés et installés, il reste à créer un périphérique virtuel Android (Android Virtual Device) via l'onglet "Virtual Devices" en cliquant sur "New". Il suffit de lui donner un nom et de choisir comme cible "Android 2.1 - API Level 7".

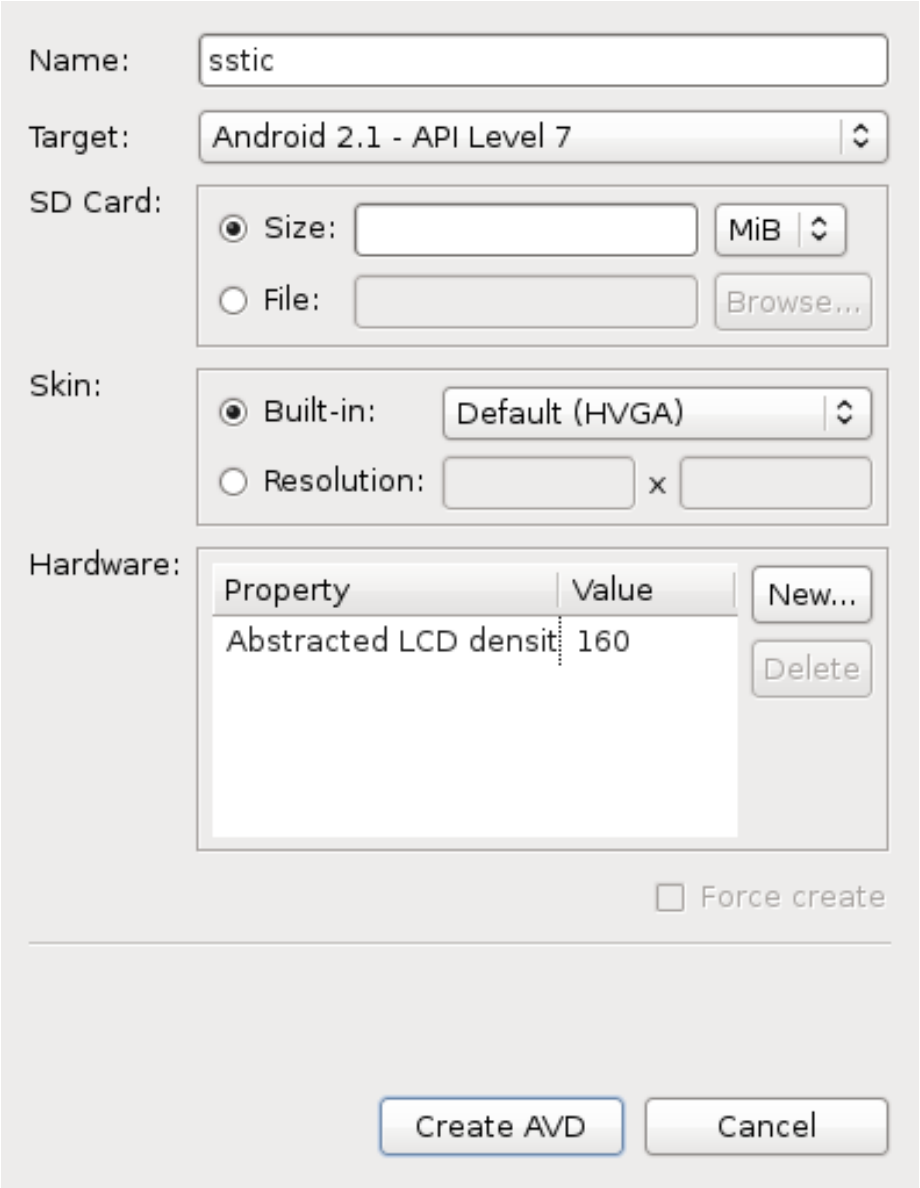

FIGURE 4 – Création d'un Périphérique Virtuel Android (AVD)

L'AVD est prêt à être lancé en cliquant sur "Start...". Le résultat est présenté ci-dessous.

FIGURE 5 – L'émulateur Android lancé

4.2 Installation des APK dans l'émulateur

L'installation des 2 APK (**com.anssi.secret.apk** et **com.anssi.textviewer.apk**) en utilisant l'outil **adb** présent dans le répertoire **tools** du SDK :

```
$ cd tools
$ ./tools/adb start-server
$ ./tools/adb install com.anssi.textviewer.apk
191 KB/s (16372 bytes in 0.083s)
        pkg: /data/local/tmp/com.anssi.textviewer.apk
Success
$ ./tools/adb install com.anssi.secret.apk
168 KB/s (19809 bytes in 0.114s)
        pkg: /data/local/tmp/com.anssi.secret.apk
Success
```

Les 2 applications sont maintenant disponibles sur le téléphone.

FIGURE 6 – Applications Secret et TextViewer installées dans l'émulateur

4.3 Interactions avec les applications

4.3.1 TextViewer

Une fois lancée, l'application TextViewer affiche simplement le contenu du fichier **chiffre.txt** contenu dans l'APK.

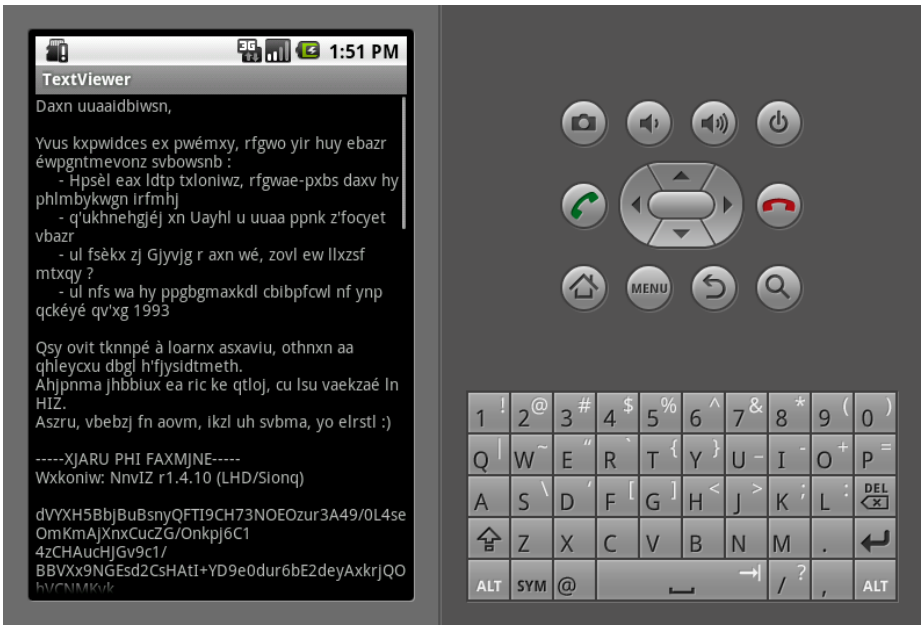

FIGURE 7 – L'application TextViewer une fois lancée

Le travail sur le fichier est détaillé dans les trois sections suivantes. Il se déroule en deux étapes :

- une première assez simple consistant dans le déchiffrement du texte. Elle permet d'accéder à un mode d'emploi de l'application Secret.
- une seconde étape de crypto discutée par la suite

Il est réalisable complètement en dehors de l'émulateur en travaillant sur le fichier **chiffre.txt**.

4.3.2 Secret

Pour ce qui est de l'application Secret, celle-ci attend la validation de 4 positions GPS et le passage d'un mot de passe.

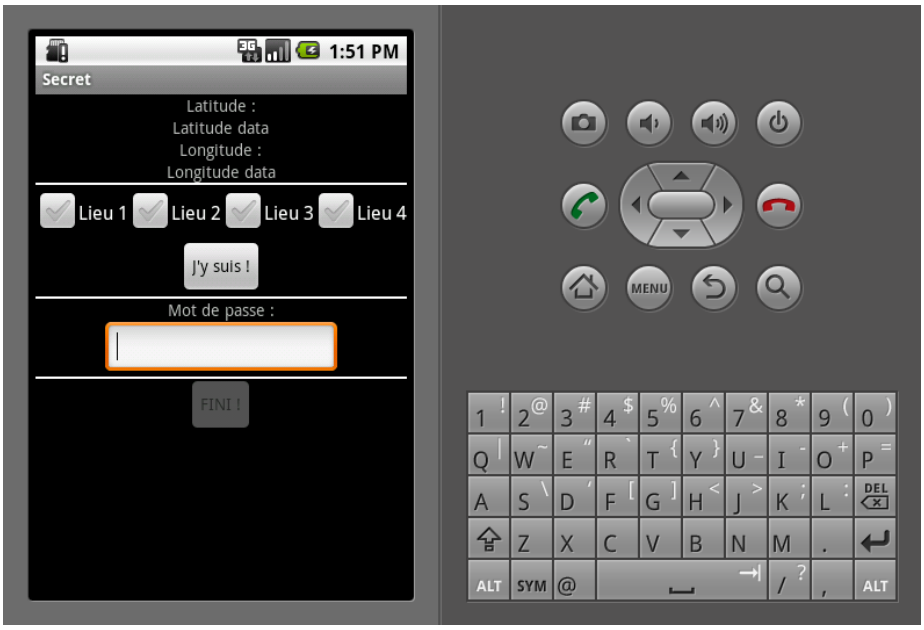

FIGURE 8 – L'application Secret lancée

L'émulateur Android autorise le passage de coordonnées GPS via sa console en écoute sur `localhost:5554`. En s'y connectant via **telnet**, la syntaxe pour changer les informations de latitude et de longitude est la suivante. Pour passer une longitude de -1.64 degrés et une latitude de 48.11 degrés :

```
$ telnet localhost 5554
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
OK
geo fix -1.64 48.11
OK
```

Les changements sont directement pris en compte dans l'application. Ceci couvre les interactions de base avec l'application.

4.4 Utiliser GDB avec l'émulateur Android

Pour faciliter la compréhension du fonctionnement interne de Secret en parallèle avec le travail de reverse sur un de ses composants (discuté par la suite), il est extrêmement utile de pouvoir debugger son fonctionnement.

La mise en oeuvre de gdb et son utilisation pour suivre le fonctionnement d'une application tournant dans l'émulateur est décrit ici. Les étapes sont les

suivantes :

- Obtention de **gdbserver** pour android et du client (via le NDK)
- Installation de **gdbserver** dans l'émulateur
- Mise en place d'une redirection de port (pour un accès hors de l'émulateur)
- Attachement à un processus dans l'émulateur
- Connection du client **gdb** hors de l'émulateur

Une fois l'archive du NDK Android téléchargée et décompressée, le binaire de **gdbserver** peut être poussé sur l'émulateur en utilisant l'utilitaire **adb** disponible dans le répertoire **tools** du SDK :

```
$ cd android-ndk-r3
$ find . -name '*gdb*'
./build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/arm-eabi-gdbtui
./build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/gdbserver
./build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/arm-eabi-gdb
./build/prebuilt/linux-x86/arm-eabi-4.2.1/bin/arm-eabi-gdbtui
./build/prebuilt/linux-x86/arm-eabi-4.2.1/bin/gdbserver
./build/prebuilt/linux-x86/arm-eabi-4.2.1/bin/arm-eabi-gdb
$ ../android-sdk-linux_86/tools/adb push \
  build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/gdbserver \
  /data/local
857 KB/s (120600 bytes in 0.137s)
```

Avant de lancer **gdbserver**, on met d'abord en place une redirection du port 1234/tcp dans l'émulateur vers le port 1234/tcp localement sur notre machine. Ceci va permettre de debugger à distance (hors de l'émulateur) une application tournant dans l'émulateur. La redirection se met en place d'une simple commande via la console de l'émulateur :

```
$ telnet localhost 5554
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
OK
redir add tcp:1234:1234
OK
```

gdbserver peut ensuite être lancé dans l'émulateur sur le processus à débugger. **ps** permet de récupérer le PID de l'application déjà lancée à laquelle **gdbserver** doit s'attacher.

```
$ ../android-sdk-linux_86/tools/adb shell
# cd /data/local
# ./gdbserver :1234 --attach 186
Attached; pid = 186
Listening on port 1234
```

Sur l'hôte, une fois le client **gdb** lancé, la commande "target remote localhost :1234" permet de se connecter via la redirection au debugger attaché à l'application dans l'émulateur.

```
$ ./build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/arm-eabi-gdb
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=x86_64-unknown-linux-gnu --target=arm-elf-linux".
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0xafe0da04 in ?? ()
(gdb)
```

5 Déchiffrement du message texte (chiffre.txt)

Le fichier **chiffre.txt** contenu dans **com.anssi.textviewer.apk** et affiché au lancement de l'application contient le texte suivant. Celui-ci est également attaché à ce document.

```
Daxn uuaaidbiwsn,

Yvus kxpwidces ex pwémxy, rfgwo yir huy ebazr éwpgntmevonz svbowsnb :
        - Hpsèl eax ldtp txloniwz, rfgwae-pxbs daxv hy phlmbykwgn irfmhj
        - q'ukhnehgjéj xn Uayhl u uuaa ppnk z'focyet vbazr
        - ul fsèkx zj Gjyvjg r axn wé, zovl ew llxzsf mtxqy ?
        - ul nfs wa hy ppgbgmaxkdl cbibpfcwl nf ynp qckéyé qv'xg 1993

Qsy ovit tknnpé à loarnx asxaviu, othnxn aa qhleycxu dbgl h'fjysidtmeth.
Ahjpnma jhbbiux ea ric ke qtloj, cu lsu vaekzaé 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+96HO54cyHZTGkO3I1QO5dbP2YUPf0V5Z8P/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/Qa59vOfxe4ZdAah7xXBhiFAPVw5fmZ9FOXzZI2
G1s5K5u2rnfZnRdl/KwZZ8gEVcBRIsJJUqmVKgvlMOp5KuxpQwDXNgv/4LyBnyfO
xMaD7gOpVPvxIuexuCQ7SFPoU7Xgqr3FhnlNd+CvN2bjXnl/PZgfAYpxufJg4jNd
8nI+8qvXVKkLBypUuM/zrb8Z/2CWYrXdsuf97OHcvVgtz/KriKPuQXvfSK9pUnDA
DIdqhidda2WMszboUGkd4LYhn06WmE2+QVzqkX/nUOR+ccX5HveRA64b1PfdZm0f
bKQtzo6pT5HGC6U3Fwx2r4dRDaC95+ejCKxXb3Aefnw2n7HIzT3iWPYBtk6oKAT7
vsMOU/45/OvDMnW4GVnAb5OxNRf5BH3VTVsOqI13gNb3cIFSXVtfXOfAB656Lv5U
A73RBF43ALXe7uegYLFrEyHjJwcRVFSjmcGGSbHCcah0MCEOOXP1ToiVVo6xTmDt
SYqEgjlhpR4iqVx1Z+JUsXnk2CqX+u7rXY5DkTZ8xahLuTyXQ7JgZAEfc29vufQl
JPsuefYofQAsQH5ouSWfIN9tZPeqEM50kNQ+jZvAJrNJADvYWpop+8rCHpFBHKLl
NBTZYbeRIwNVUdZCJnkLVpIBUpLCIzXYK4UJJglJyd6MKfguSeWKUaLkHJNQs5bO
JKQCLd1cdwu94jCzwNOWtCvDUmv6m0yzzHvYVka9z/jRz9lqvJXJGYdn+8kRUTZz
Xe3aRrl7+cZOG2nNjmIz1URMlqTatyCaX3ww+rJTVlmUHohMAOxzI7eV4RNpj39P
UpHRNqXOF4tykU9cCWs9/qvtudsi7l2HaoXfbSCAHLTs4NkaK3u6ft3PnPgtoqJt
YQEk90G5QbHv9Nf1J0eb9B5TtZhE3OMp8MZmfywaKfHDCJJWnbW4d1JelEMP0k27
OnSMrZ84G6nT5vW36ZFWjnKZHl6Uyof8LRtrSIbeGROSpY4wLOZavMqX8zmHobo0
Q7UTtrXSLITZ8BX/cF89KBXukj/qMRWJAruiJLyM7iKx/mdBO2uikuH/IfLXEXWB
hsin7IbLoMub4Ejc95ypJKBXoWqJMpMDYZPAEpNYX6X7hXIcWTQOUS4OHABDVNG+
BxkDEH5ZQJU7JRDiotaCuHvykEEbyfZF4WElle91aaLmWXGbkOtWot0eUzVJh/cv
GBKhbbN=
=8CVr
-----EOW ICU JDILJV DAD VUVCL-----
```

Le texte est illisible mais conserve sa structure : certains caractères n'ont donc pas été modifiés par le mode de chiffrement utilisé. C'est notamment le cas des espace mais également de celui des tirets ('-') et plus généralement de l'ensemble des signes de ponctuation et des caractères accentués.

Deux blocs rappelant le format PEM d'encodage des certificats X.509 ou les versions "ASCII armored" des messages/clés GPG semblent présents dans le texte. Il s'agit plus vraisemblablement d'éléments GPG, l'entête du premier message coincidant parfaitement au niveau format avec un celle d'un message GPG classique :

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

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

Cette remarque est également valable pour les marqueurs de fin

```
Classique  : -----END PGP MESSAGE-----
chiffre.txt: -----LNE IZL RYBZAHX-----
```

Cette conclusion s'applique également au second message dont le format colle parfaitement avec celui d'une clé publique GPG :

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

             ...

             -----END PGP PUBLIC KEY BLOCK-----

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

             ...

             -----EOW ICU JDILJV DAD VUVCL-----
```

Il est intéressant de noter que le mode de chiffrement utilisé n'est pas une simple permutation alphabétique (du type rot13) car des éléments identiques du texte initial n'ont pas le même chiffré en fonction de leur position dans le texte. C'est le cas notamment de la ligne se trouvant sous les entêtes des élément GPG :

```
Habituel       : Version: GnuPG v1.4.10 (GNU/Linux)
Elément GPG #1 : Wxkoniw: NnvIZ r1.4.10 (LHD/Sionq)
Elément GPG #2 : Ayazipg: ZjzJP c1.4.10 (GON/Eesog)
```

Une idée qui vient à l'esprit consiste donc à "soustraire" des éléments clairs connus du message avec leurs équivalents chiffrés. Pour cela, il est possible d'utiliser les entêtes et marqueurs de fin des éléments GPG. Différentes soustractions sont à tester sur les valeurs ASCII des caractères : XOR, soustraction avec différentes valeurs de modulo ... Après quelques essais, on tombe rapidement sur ceci :

```
>>> s1="BEGINPGPPUBLICKEYBLOCK"
>>> s2="CXZESJPWPVUEEHENFBMHVG"
>>> for k in range(len(s1)):
...   dec = (ord(s2[k]) - ord(s1[k])) % 26
...   print "%02d | %s" % (dec, '#'*dec)
...
01 | #
19 | ###################
19 | ###################
22 | ######################
05 | #####
20 | ####################
09 | #########
07 | #######
00 |
01 | #
19 | ###################
19 | ###################
22 | ######################
05 | #####
20 | ####################
09 | #########
07 | #######
00 |
01 | #
19 | ###################
19 | ###################
22 | ######################
```

Un motif apparait dans le résultat de cette soustraction un à un des caractères modulo 26 : [1, 19, 19, 22, 5, 20, 9, 7, 0]. Pour déchiffrer le message, il suffit donc de cycler sur le tableau de décalages obtenu pour les caractères alpha du message. Ce qui donne en Python :

```
>>> txt = open("chiffre.txt").read()
>>> def uncipher_text(txt, pwd):
...     i = 0
...     res = ''
...     for c in txt:
...         if not c.isalpha():
...             res += c
...             continue
...         dec = pwd[i]
...         i = (i + 1) % len(p)
...         tmp = ord(c) - dec
...         if ((c.isupper() and tmp < ord('A')) or
...             (not c.isupper() and tmp < ord('a'))):
```

```
...                     tmp += 26
...             res += chr(tmp)
...         return res
...
>>> print uncipher_text(txt, [1, 19, 19, 22, 5, 20, 9, 7, 0])

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+96CO54ipAZSNrS3D1WF5wbO2FBTa0B5Q8I/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/FcQS8gDCjFMOjCJTxtZFmmeMOo5RbbkWnWXMnc/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/dwBO2tpryC/OwEXDEDF
cyzg7IaSvQph4Vcc95xwQOWDfPqITwQYEQIADwUCS6D7yQIbDAUJAJ40AAAKCRB+
HodDDO5GUEA7AKDhvaeXaYoyjLLftlQY4WDssi91vgCfWWNio0oCfm0eTgCNc/im
ZBJoifI=
=8IMk
-----END PGP PUBLIC KEY BLOCK-----


>>>
```

La version déchiffrée du fichier est également attachée à ce document.

La suite du challenge consiste donc à fournir à l'application Secret les coordonnées GPS de 4 lieux (associés aux 4 énigmes) et un mot de passe (à extraire des 2 éléments GPG présent dans le message).

6 Réponses aux 4 questions

6.1 Après les rump sessions, rendez-vous chez ce galliforme breton

Cette année comme l'année précédente, le social event du SSTIC se déroule au Coq Gabdy, 156 rue d'Antrain, à Rennes. Les premières coordonnées GPS sont donc offertes :

- Latitude : 48.123
- Longitude : -1.667

6.2 L'abandonnée de Naxos y part pour d'autres cieux

Selon Wikipedia, Naxos est une île grecque de la mer Egée. Elle "doit en partie sa célébrité à la mythologie : selon la légende, Thésée y abandonna Ariane ...". Le pas de tir d'Ariane à Kourou en Guyane se trouve aux coordonnées GPS suivantes (également offertes) :

- Latitude : 5.15865
- Longitude : -52.650261

6.3 Le frère de Marvin y est né, sous la grosse table ?

La réponse à cette question n'a été trouvée par l'auteur. Des coordonnées valides ont malgré tout été obtenues via le reverse de la bibliothèque libhello-jni.so :

- Latitude : 37.88
- Longitude : -122.3

Les coordonnées placent la réponse dans les environs de San Francisco.

6.4 Le nez de ce gigantesque capitaine ne fut libéré qu'en 1993

La voie El nose d'El Capitan dans la vallée de Yosémite aux USA fut escaladée en libre par Lynn Hill en 1993. Les coordonnées GPS associées sont les suivantes :

- Latitude : 37.734
- Longitude : -119.63737.73

6.5 Note

Après la phase de reverse de la bibliothèque libhello-jni.so décrite plus loin dans le document, les informations suivantes sur les coordonnées sont disponibles :

- Les 8 éléments de coordonnées (4 couples de latitudes et longitudes) sont initialement multipliés par π avant d'être passés à la fonction de dérivation de clé deriverclef() de la bibliothèque libhello-jni.so.
- Les 8 éléments de coordonnées sont arrondis et convertis en entier. Le résultat final constitue un tableau de 8 octets.
- Le tableau de 8 octets est ensuite utilisé pour reconstruire le nom d'une classe utilisée par la suite dans le programme. Une valeur invalide d'un élément de coordonées prévient l'accès à cet classe.
- Les 4 premiers éléments de coordonnées (associés aux 2 questions faciles) sont utilisés dans la procédure de dérivation de clé. Les 4 derniers ne le sont pas.

Plus de détails sont donnés dans la section discutant le reverse de la bibliothèque.

7 Obtention du mot de passe chiffré en GPG

7.1 Introduction

Le mot de passe demandé par l'application Secret après le passage des coordonnées GPS est chiffré en GPG. On commence donc par analyser les éléments GPG fournis.

Un élément GPG (clé publique, clé privée, message chiffré, ...) est un flux de données qui se décompose en un ensemble de paquets de données. Chaque paquet possède un tag qui identifie son type, une longueur et des données (spécifiques à son type). Chaque élément GPG possède une structure particulière fait d'un ensemble de ces paquets. Le format des éléments GPG et des paquets est documenté dans la RFC4880.

Un élément GPG est donc initialement un blob binaire. Un encodage ASCII (ASCII Armor) de ce blob binaire est défini : L'élément GPG est converti en base64, se voit concaténer une somme de contrôle (sur 3 octets, calculée avant le passage en base64) et est ensuite mis entre un entête et un marqueur de fin. Il s'agit du format des 2 éléments GPG présent dans le message.

7.2 Analyse de la clé publique GPG

La clé GPG a été sauvée dans un fichier **pubkey.pgp**. On commence par l'analyser dans les détails.

L'option `--list-packets` de **gnupg** permet de réaliser une première analyse de la clé. Les valeurs numériques (algorithmes, fonctionalités, ...) retournées par la commande ci-dessous sont documentées dans la RFC4880.

```
$ gpg --list-packets pubkey.gpg
:public key packet:
        version 4, algo 17, created 1268841417, expires 0
        pkey[0]: [1024 bits]
        pkey[1]: [160 bits]
        pkey[2]: [1023 bits]
        pkey[3]: [1022 bits]
:user ID packet: "Personne <invalide@nomdedomaine.tld>"
:signature packet: algo 17, keyid 7E1E87430CEE4650
        version 4, created 1268841417, md5len 0, sigclass 0x13
        digest algo 2, begin of digest ce 3e
        hashed subpkt 2 len 4 (sig created 2010-03-17)
        hashed subpkt 27 len 1 (key flags: 03)
        hashed subpkt 9 len 4 (key expires after 120d0h0m)
        hashed subpkt 11 len 5 (pref-sym-algos: 9 8 7 3 2)
        hashed subpkt 21 len 5 (pref-hash-algos: 8 2 9 10 11)
        hashed subpkt 22 len 2 (pref-zip-algos: 2 1)
        hashed subpkt 30 len 1 (features: 01)
        hashed subpkt 23 len 1 (key server preferences: 80)
        subpkt 16 len 8 (issuer key ID 7E1E87430CEE4650)
```

```
        data: [157 bits]
        data: [160 bits]
:public sub key packet:
        version 4, algo 16, created 1268841417, expires 0
        pkey[0]: [1024 bits]
        pkey[1]: [3 bits]
        pkey[2]: [1024 bits]
:signature packet: algo 17, keyid 7E1E87430CEE4650
        version 4, created 1268841417, md5len 0, sigclass 0x18
        digest algo 2, begin of digest 40 3b
        hashed subpkt 2 len 4 (sig created 2010-03-17)
        hashed subpkt 27 len 1 (key flags: 0C)
        hashed subpkt 9 len 4 (key expires after 120d0h0m)
        subpkt 16 len 8 (issuer key ID 7E1E87430CEE4650)
        data: [160 bits]
        data: [159 bits]
```

Le premier paquet correspond à la clé publique : une clé DSA 1024 bits. Le second paquet contient l'identité du possesseur de la clé (User ID) : "Personne <invalide@nomdedomaine.tld>".

Le paquet suivant est la signature de l'ID et de la clé publique précédente (avec la clé privée associée). Le paquet de signature contient des informations de préférence, notamment sur les algorithmes crypto et de compression qui peuvent être utilisés lors de l'échange de données avec GPG. Notamment :

- **Algorithmes de chiffrement symétrique** : AES256 (9), AES192 (8), AES128 (7), CAST5 (3), 3DES-EDE (2)
- **Algorithmes de hash** : SHA256 (8), SAH-1 (2), SHA384 (9), SHA512 (10), SHA224 (11)
- **Algorithmes de compression** : ZLIB (2), ZIP(1)

La clé principale de l'utilisateur étant une clé DSA, elle n'est utilisable que pour des opérations de signature. Le paquet suivant contient une sous-clé ElGamal 1024 bits utilisable pour les opérations de chiffrement. Elle est signée avec la clé DSA de l'utilisateur (dernier paquet). Les éléments notés `pkey[0]`, `pkey[1]` et `pkey[2]` correspondent respectivement au nombre premier p (1024 bits), au générateur du groupe g (3 bits) et à la partie publique $y = g^x \ mod \ p$ (avec x la partie secrète).

L'extraction des 3 éléments de la clé publique ElGamal se fait assez simplement avec les informations de la RFC4880 (à la main) ou en utilisant **pgpdump** :

```
$ pgpdump -ilmp pubkey.gpg
...
Old: Public Subkey Packet(tag 14)(269 bytes)
   Ver 4 - new
   Public key creation time - Wed Mar 17 16:56:57 CET 2010
```

```
   Pub alg - ElGamal Encrypt-Only(pub 16)
   ElGamal p(1024 bits) - ee 0e fd ed 46 ab 3b fa 30 c9 33 67 86 8e 53 e0
                          d5 83 12 72 34 f3 69 f1 af 0f 79 e9 fa b2 00 0a
                          98 40 62 18 4b 1f 47 9d 0f b9 0c e0 a3 68 ab 7f
                          56 62 43 48 1a a5 b4 33 86 9c 7c 0f 6d 55 6a fd
                          ff 17 31 a6 38 db ee 5d 86 86 f0 5a 84 95 40 19
                          20 30 e0 8a ab 0f 7b ba 79 ad d6 ac a9 a4 86 a2
                          1a 15 46 6a f7 45 f9 25 b1 9c f5 48 f5 13 49 2c
                          f5 4e 53 b0 6a 08 dc 94 a0 f0 56 65 99 f6 af 43
   ElGamal g(3 bits) - 05
   ElGamal y(1024 bits) - 8a 13 78 c3 52 5d b0 b4 0a d2 ad bb 16 04 8b c8
                          6f 38 2b a8 99 e6 63 f7 e9 91 03 aa b1 45 61 ee
                          94 c6 f9 bc 3c cc e2 2c 86 a5 35 52 58 81 1e 30
                          29 51 9e a9 28 50 f3 39 4e bd f8 f4 5b b2 cc b6
                          a1 19 3c 36 50 f1 45 ff 6c cf 3d 44 54 9a 6d cf
                          ea 2d 80 cd 57 19 5b 24 a7 d3 ee 61 5d fd dc 01
                          d3 6b 69 af 20 bf 3b 01 17 0c 40 c5 73 2c e0 ec
                          86 92 bd 0a 61 e1 57 1c f7 9c 70 40 e5 83 7c fa
Old: Signature Packet(tag 2)(79 bytes)
        Ver 4 - new
...
```

7.3 Analyse du message chiffré en GPG

Le message chiffré en GPG contient les éléments suivants :

```
$ gpg --list-packet msg.gpg
:pubkey enc packet: version 3, algo 16, keyid 905AA88C96C9EAD0
        data: [1021 bits]
        data: [1021 bits]
:pubkey enc packet: version 3, algo 1, keyid 2A9C6105E1F67BBD
        data: [1021 bits]
:encrypted data packet:
        length: 60
gpg: encrypted with RSA key, ID E1F67BBD
gpg: encrypted with 1024-bit ELG-E key, ID 96C9EAD0, created 2010-03-17
      "Personne <invalide@nomdedomaine.tld>"
gpg: decryption failed: secret key not available
```

Les données du messages sont présentes dans le dernier paquet, chiffrées avec un algorithme de chiffrement symétrique (disponible uniquement au destinataire du messages), via une clé de session générée pour l'occasion. Le premier paquet du message contient la clé de session chiffrée en ElGamal avec la partie publique de Persone. Le second paquet contient cette même clé de session chiffrée en RSA. L'utilisateur (ID 0x2A9C6105E1F67BBD) à qui cette clé appartient est inconnu.

Les 2 éléments de 1021 bits (qui constituent le chiffré ElGamal) dans le premier paquet du message sont respectivement $c_1 = g^k \ mod \ p$ et $c_2 = m \cdot y^k \ mod \ p$ avec :

- g le générateur du groupe (i.e. 5) donné plus haut,
- y la partie publique ElGamal de personne
- p le modulo
- k la clé éphémère ElGamal utilisée pour le chiffrement de la clé de session.

Ces paramètres dont nous aurons besoin peuvent encore une fois être extraits à la main ou en utilisant pgpdump :

```
$ pgpdump -ilmp msg.gpg
Old: Public-Key Encrypted Session Key Packet(tag 1)(270 bytes)
        New version(3)
        Key ID - 0x905AA88C96C9EAD0
        Pub alg - ElGamal Encrypt-Only(pub 16)

   ElGamal g^k mod p(1021 bits) - 18 2e f7 4c 55 ce c9 bc b7 13 8f 7f
                                  d0 6e 32 bc 79 89 b4 79 d2 b6 85 6e
                                  6c 63 7f 4a 2a a0 73 a0 b5 e3 22 4e
                                  12 98 98 08 6b bd 8f 5f c8 15 00 e8
                                  f4 61 83 ce 4d 86 9c d4 66 23 e5 ca
                                  f6 5d 21 a7 1e ac 5f 67 5d 7c 76 df
                                  c5 a2 4e 80 22 51 1d 09 9d b2 c8 77
                                  b8 81 97 af 8a 49 3b 84 09 f4 50 ae
                                  80 e4 f9 00 cf 23 55 70 f6 f9 18 fa
                                  93 d0 55 0b c0 bc 2c e4 5c 2a 14 f1
                                  f6 e5 06 c7 d0 20 da 5c

   ElGamal m * y^k mod p(1021 bits)- 1e e2 e1 4d b9 ed 60 ac 65 bb 85
                                  e4 5a 98 de a3 82 97 d1 0b 95 32 65
                                  f7 8a 9b d3 cf ba d9 f2 44 bf 96 81
                                  7a 9c b1 78 b4 1c 17 de e4 45 9e ba
                                  2f 63 35 20 d3 78 ef 8f 03 5a ad 2f
                                  33 55 01 ca 5d 09 8e 94 dd 6b 55 78
                                  3a 73 de 1d c5 05 d9 9d 03 99 48 2e
                                  62 b8 86 e5 7d 05 20 50 57 8b 7f 22
                                  78 42 d8 da 94 f7 2e e2 18 5b 09 0f
                                  87 9b 95 3b 16 bf 91 5b 16 68 89 22
                                  75 12 39 67 77 9a 2f fd 27
...
```

7.4 Tentatives infructueuses

Avant de passer à la solution décrite ci-dessous (finalement assez logique après coup), de nombreuses tentatives infructueuses ont été tentées par l'auteur.

La recherche de la partie privée de la clé ElGamal dans la copie mémoire n'a pas donné de résultat.

Une seconde tentative consistant à rechercher la clé de session utilisée pour chiffrer les données du message dans la copie mémoire (pour AES128 et AES256) n'a pas non plus donné de résultat.

Ces deux tentatives restants infructueuses, il semblait logique de penser que le bloc contenant la clé de session chiffrée en RSA n'avait pas été mis là par hasard. Au final, il n'a pas été possible de trouver son rôle dans le challenge.

Il est à noter que l'angle d'attaque décrit ci-dessous résultant du simple test des paramètres de la clé semblait **initialement** trop simple et trop peu "ludique" après la phase de déchiffrement du message **chiffre.txt**.

7.5 Une solution

7.5.1 Analyse des paramètres ElGamal

Dans le chiffrement du message par GPG, ElGamal a été utilisé pour protéger une clé de session associée à un algorithme de chiffrement symétrique protégeant les données du message (les préférences de la clé de l'utilisateur suggère un AES256 mais cette information est protégée avec la clé de session).

Le premier test à effectuer concernant les paramètres ElGamal de la clé publique est la vérification de la primalité de p, le modulo.

```
>>> from numbthy import *
>>> p = 16717040734836875527457119871084615500585219745115\
    83049992822575938062496902354614005894754053843815434 5\
    14562017713115541670130868204873941679302086020150665 4\
    67696382354201762321373114741156399995032048374486383 4\
    61432948688649530634732561502481575243565093945455695 3\
    10126433916459420936954286595494713128835 23
>>> isprime(p)
True
```

p étant premier, l'ordre du groupe est $n = p-1$. Vérifions si p est un premier sûr ("safe prime) en s'assurant que $n/2$ est lui aussi premier.

```
>>> isprime((p-1)/2)
False
```

Etonnant. Essayons donc de factoriser n. Pour cela, **PARI/GP** est un outil plus adapté (i.e. efficace) que Python.

```
$ gp
? default(lines, 1000)            # don't limit output size
%1 = 1000

? factorint(16717040734836875527457119871084615500585219745115830499928225759380624969023546140058947540538438154345145620\
            17713115541670130868204873941679302086020150665467696382354201762321373114741156399995032048374486383461432948\
            68864953063473256150248157524356509394545569531012643391645942093695428659549471312883522)
```

```
%2 =
[2 1]

[1218055055968339 1]
[1263847861201609 1]
[1271483404519507 1]
[1306620742471661 1]
[1435469233657999 1]
[1436852757281407 1]
[1455144603998677 1]
[1593684693149279 1]
[1724498562415303 1]
[1780716924867173 1]
[1917204589315909 1]
[1922550339910303 1]
[1975985172968039 1]
[2077649398994551 1]
[2108107767794563 1]
[2132773087614569 1]
[2133463604190461 1]
[2174110522001753 1]
[2227343475745711 1]
[3165493139633045911 1]

?
```

La sortie est constituée de la liste des facteurs premiers de n avec leur exposant associé dans la décomposition. Ici chaque facteur n'apparait qu'une fois dans la décomposition, i.e. tous ont un exposant à 1.

factorint() utilise une combinaison de méthodes pour la factorisation (voir le manuel de PARI/GP). La primalité des valeurs retournée n'est pas garantie mais peu être testée avec **isprime()**. C'est bien le cas des valeurs obtenues.

Les valeurs obtenues sont toutes de taille comparable (51 bits) sauf pour la dernière, légèrement supérieure (62 bits) :

```
>>> import math
>>> d= [ 2, 1218055055968339, 1263847861201609, 1271483404519507,
            1306620742471661, 1435469233657999, 1436852757281407,
            1455144603998677, 1593684693149279, 1724498562415303,
            1780716924867173, 1917204589315909, 1922550339910303,
            1975985172968039, 2077649398994551, 2108107767794563,
            2132773087614569, 2133463604190461, 2174110522001753,
            2227343475745711, 3165493139633045911 ]
>>> for c in d:
...     print math.log(c,2)
...
```

```
1.0
50.1135007677
50.1667442293
50.1754340553
50.214761871
50.350443833
50.3518336513
50.3700839504
50.501287647
50.6150983494
50.6613796172
50.7679257218
50.7719427974
50.8114935448
50.8838736449
50.9048700436
50.9216519043
50.9221189224
50.9493467057
50.9842454747
61.4571359769
```

L'existence d'une décomposition de l'ordre du groupe en facteurs premiers de taille raisonnable permet d'envisager l'utilisation de l'algorithme de Pohlig-Hellman pour le calcul du logarithme discret :

- soit pour retrouver la valeur de x, la clé privée Elgamal, i.e. trouver x tel que $y = g^x \ mod \ p$
- soit pour retrouver la valeur de k, la clé éphémère ElGamal, i.e. trouver k tel que $c_1 = g^k \ mod \ p$

L'obtention de l'une ou l'autre des 2 valeurs permet de remonter à m.

L'algorithme de Pohlig-Hellman permet de ramener le calcul du logarithme discret dans Z/pZ au calcul de logarithmes discrets dans des sous-groupes d'ordres plus petits (de la taille des premiers précédents) puis à une résolution d'un système de congruence en utilisant le Théorème des Restes Chinois. Le calcul du logarithme discret dans les sous-groupes peut-être réalisé en utilisant l'algorithme rho de Pollard.

Le lecteur intéressé trouvera une bonne description de l'algorithme de Pohlig-Hellman en 3.6.6 dans le chapitre 3 du Handbook of Applied Cryptography de Menezes, van Oorschot et Vandstone (téléchargeable gratuitement).

Pour ce qui nous concerne, la complexité de l'attaque est liée au calcul du logarithme discret dans le sous-groupe associé au plus grand des facteurs. Elle est de l'ordre de $O(\sqrt{3165493139633045911}) = O(2^{31})$.

La section suivante décrit la mise en oeuvre de l'attaque, i.e. les outils utilisés pour obtenir l'une des 2 valeurs précédentes.

7.5.2 Modifications de l'implémentation de Pohllig-Hellman de LiDIA

Pour la mise en oeuvre de l'attaque, nous utilisons une version modifiée de l'implémentation de pohlig-hellman (**dlp_appl**) disponible dans LiDIA.

L'outil **dlp_appl** disponible dans les exemples de la librairie prend en paramètre a, b et p et retourne x tel que $b = a^x \ mod \ p$. Pour cela, il tente de réaliser lui-même la factorisation de $p-1$. Malheureusement, le progamme impose une limite (inférieure à la valeur de p) sur les nombres qu'il accepte de factoriser.

Ayant déjà accès à la factorisation de $p-1$ (via **PARI/GP**), la première étape consiste donc à modifier rapidement le programme pour qu'il prenne en entrée l'ensemble de facteurs. Le patch associé (**use_factors.patch**) est attaché à ce document.

```
$ DIR=http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/ftp/LiDIA/current/
$ wget -q ${DIR}/lidia-base-2.3.0.tar.gz
$ wget -q ${DIR}/lidia-FF-2.3.0.tar.gz
$ tar xfz lidia-base-2.3.0.tar.gz
$ tar xfz lidia-FF-2.3.0.tar.gz
$ cd lidia-2.3.0
$ patch -p1 -i /tmp/use_factors.patch
patching file src/finite_fields/discrete_log/pohlig_hellman/dlp.cc
patching file src/finite_fields/include/LiDIA/dlp.h
patching file src/finite_fields/discrete_log/pohlig_hellman/dlp_appl.cc
$ ./configure
$ make
$ cd examples/finite_fields
$ make dlp_appl
```

Il suffit ensuite de lancer le programme en lui passant les a et b précédents directement suivis de la liste des facteurs premiers de $p-1$. Dans notre cas :

```
$ ./dlp_appl a b p1 ... p21
```

retournera x tq $b = a^x \ mod \ (p1 \cdot p2 \cdot \ \ldots \ \cdot p20 \cdot p21 + 1)$

7.5.3 Obtention de x et/ou k

Comme évoqué précédemment, l'obtention d'une seule des 2 valeurs x ou k est suffisante pour remonter au message. Malgré tout, la complexité de l'algorithme évoquée précédemment est une borne supérieure. Des valeurs spécifiques de x et k vont dépendre le temps de calcul : il est possible qu'un des 2 calculs de logarithme discret aboutisse bien avant le premier.

Dans les faits, les 2 calculs ont été réalisés en parallèle sur la même machine (Intel Xeon multicoeur à 3GHz). Le calcul de k a pris environ 7 heures. Celui de x environ 3 fois plus de temps, i.e. 21 heures.

Pour le calcul de k, les résultats intermédiaires de l'algorithme de Pohlig Hellman sont les suivants :

- $1 \ mod \ 2$

- 872369050350763 *mod* 1218055055968339
- 1068041787979718 *mod* 1263847861201609
- 958955728833198 *mod* 1271483404519507
- 408764087210986 *mod* 1306620742471661
- 571880503612888 *mod* 1435469233657999
- 1151793512882267 *mod* 1436852757281407
- 715557480710616 *mod* 1455144603998677
- 1064806456619047 *mod* 1593684693149279
- 1244667582116960 *mod* 1724498562415303
- 1261679303781737 *mod* 1780716924867173
- 633643813923731 *mod* 1917204589315909
- 793611590273435 *mod* 1922550339910303
- 543338362966620 *mod* 1975985172968039
- 1664299917442430 *mod* 2077649398994551
- 326601813387782 *mod* 2108107767794563
- 1241658938313770 *mod* 2132773087614569
- 826483947408468 *mod* 2133463604190461
- 1041266915504746 *mod* 2174110522001753
- 946740799489959 *mod* 2227343475745711
- 859608165945422236 *mod* 3165493139633045911

Le résultat final est :

```
k = 3442798784842268266210142027076483000991667721152362526627\
    90990317493690573
```

Pour le calcul de x, les résultats intermédiaires de l'algorithme de Pohllig Hellman sont les suivants :

- 1 *mod* 2
- 978262958165838 *mod* 1218055055968339
- 624268583195909 *mod* 1263847861201609
- 1129104943338998 *mod* 1271483404519507
- 450980944852767 *mod* 1306620742471661
- 614354085824131 *mod* 1435469233657999
- 388393936805862 *mod* 1436852757281407
- 210036770245526 *mod* 1455144603998677
- 1480527190825160 *mod* 1593684693149279
- 598027909965540 *mod* 1724498562415303
- 8657386810826 *mod* 1780716924867173
- 1397308102252136 *mod* 1917204589315909
- 868725239079599 *mod* 1922550339910303
- 243798483328639 *mod* 1975985172968039
- 1749977111909658 *mod* 2077649398994551
- 1258511240243730 *mod* 2108107767794563
- 1851179581442108 *mod* 2132773087614569

- 1648685792733093 *mod* 2133463604190461
- 738043239287262 *mod* 2174110522001753
- 923697751917423 *mod* 2227343475745711
- 1458660248217431679 *mod* 3165493139633045911

Le résultat final est :

```
x = 821600028463037887808776457389953896934820565669575595803 6\
    937166108823945599870897923017427049218488709352775647379 3\
    037846785162980029537236246269728347946278173732394323079 9\
    060590576564545970261610205470 7
```

7.5.4 Obtention du mot de passe

k ayant été le premier résultat obtenu, on l'utilise pour remonter au message. On commence par vérifier que le calcul précédent a bien donné le bon résultat :

```
>>> from numbthy import *
>>> g = 5
>>> p = 167170407348368755274571198710846155005852197451158304999282257\
       593806249690235461400589475405384381543451456201771311554167013\
       086820487394167930208602015066546769638235420176232137311474115\
       639999503204837448638346143294868864953063473256150248157524356\
       509394545569531012643391645942093695428659549471312883523L
>>> k = 344279878484226826621014202707648300099166772115236252662790990\
       317493690573L
>>> c1 = powmod(g, k, p)
>>> c1
169822038142468984697137543358043178114004389893275776026472655124717 84\
947823252861506621706312117314961504678001892826821718760679350012217 85\
054096458791526235220046184664501846975379398566888259427176847525827 48\
953969034647898497742072399486033096215911879504340060720327725553178 07\
05693335556362425341398 0L
```

Il s'agit bien de c_1. Maintenant :

$c_2 = m \cdot y^k \ mod\ p = m \cdot (g^x)^k \ mod\ p = m \cdot (g^k)^x \ mod\ p = m \cdot h^k \ mod\ p$

on a donc

$$m = c_2 / h^k \ mod\ p$$

Ce qui s'écrit en Python (le module modinverse est attaché à ce document) :

```
>>> from Crypto.PublicKey.RSA import number  # for long to bytes conversion
>>> from modinverse import modinv
>>> c2 = 216890625916035677878670951228568285058525799824314598453768 76\
         580006150881227075506616799539606680834724530234890556474227 79\
         452652922174552040991110236043979548359198566749373437441727 14\
         011617250373532355032301197843872108846259364830548290956071 96\
```

```
        062234991495574936237939154025934198593104188136393953770791L
>>> y = 969603077155026238418838220337761808744284685743631190487303704\
        838583042127279584086006520440405325271147188479200679591833075\
        311173665377778810859662005837225637283981588924689861041853322\
        149675291941024141624328796962731433945830003011996287050891849\
        529871343532711638193453809545679304797908744480365928 90L
>>> k = 344279878484226826621014202707648300099166772115236252662790990\
        317493690573L
>>> p = 167170407348368755274571198710846155005852197451158304999282257\
        593806249690235461400589475405384381543451456201771311554167013\
        086820487394167930208602015066546769638235420176232137311474115\
        639999503204837448638346143294868864953063473256150248157524356\
        509394545569531012643391645942093695428659549471312883523L
>>> m = c2 * modinv( pow(y, k, p), p) % p
>>> m
675466616815069699296930666808908814155780556964737495456057607310806 14\
036960887312549394622639204818280333846653855603384553787960283102092 90\
886613318357477144586015068485857371416859493212385918879960799234688 29\
153205324093738022691610614591347849955230198030684945191681460403290 42\
83462639957077987563L
>>> m = number.long_to_bytes(m, 128)
'\x00\x02vcom>"\x8c\x05CYt\xdd\xbc\x9c=\xf0S\x195\x10\xb9\']\xf6\xc6\xce
\xae\xa0\xb6\x18\x8c\xe9\'uuN/\x990\x8aM\xe1Xq\xe3\xac\xa4\x07\xc1\xbe
\x1c\x16\xa0\xce`\xf4#~d(\x1c\xcf\r\xa2E[\x0fn2t\xb6g\x9c]\xf4\xa5\x91`
\x1by\x82/\x0f\xe4\xd9 e2\x1c\x8d\x00\t\xdc\x95\xf6\xff\x93\xe6\xd8\xfb
\xb2\x12\xfb\xe0\x9fCFZ;et\xfd.W\x12S\x13\xdfV*\xc7\xa2-\x15\x10\xeb'
```

Joie ! Le message obtenu a bien le format attendu, i.e. celui d'un bloc type 2 PKCS#1 comme décrit en section 7.2.1 de la RFC 3447 :

```
EM = 0x00 || 0x02 || PS || 0x00 || M
```

Avec :

- EM (Encoded Message) : bloc chiffré
- PS (Padding String) : constitué d'octets aléatoires non nuls
- M (Message) : le message

```
>>> data = m.split('\x00', 2)[-1]    # extraction des données
>>> len(data)
35
```

Comme décrit dans la RFC4880, le message chiffré n'est pas composé que de la clé symétrique : celle-ci est suivie d'une somme de contrôle sur 2 octets et précédée d'un octet indiquant l'algorithme de chiffrement symétrique utilisé :

```
>>> k = data[1:-2]
>>> len(k)
```

```
32
>>> cksum = struct.unpack("!H", data[-2:])[1]
>>> cksum
4331
>>> def twocksum(s):
...     i = 0
...     for c in s:
...         i += ord(c)
...     i = i % 65536
...     return i
...
>>> twocksum(k)
4331
>>> symalg = ord(data[1])
>>> symalg
9             # i.e. AES256
```

On a donc à faire à un AES256 ce qui est cohérent avec la clé de 32 octets. Le mode d'AES utilisé par GPG pour le chiffrement des messages est spécifique : il s'agit d'une variante du mode CFB.

Il est à noter que le support de ce mode est annoncé dans PyCrypto mais ne fonctionne pas. Il est possible de l'implémenter assez rapidement en Python. Une fois ceci réalisé :

```
>>> from Crypto.Cipher import AES
>>> from pgphelper import pgp_cfb_decrypt
>>> msg = "\xdb\xff\xa8sL\x9a\xa1\x1exx\xa6\x05\xf6\x850\xba\xac
           \xbb@\x87~\x81\xbd\xf9\xa0\xbe3\xe9*-\xbc\xe0\x1c0\xd4
           #\xd5\xc0\xa7(\xd8\x10\x89\x8b'\x12\x14\xec\xba\xfe\xc9
           f0j\x96mz\x0cL\xef"
>>> dec = pgp_cfb_decrypt(AES, k, msg)
>>> dec
'\xa3\x02x\x9c[#\x97\xc4\x9e\x9bR\xa0WRQ\xe2\xbd:\x85\xc7\xdf<\xa34
09\xb2\xca\xc35 \xb8\xd0\xc2(\x97\x0b\x00\xbb,\n\xd5'
```

`pgp_cfb_decrypt()` n'ayant pas retourné None, le déchiffrement s'est bien passé (vérification des 2 octets dupliqués comme décrit en section 13.9 de la RFC4880). Le premier octet du résultat indique le type de paquet GPG : un paquet de type 8, i.e. compressé. L'octet suivant donne le type de compression utilisé, 2 pour zlib.

```
>>> import zlib
>>> d = zlib.decompress(dec[2:])
'\xac\x1eb\x07mdp.txtK\xabd\x0cO7huQcYzHEPSq82m\n'
```

On voit apparaitre le nom du fichier (mdp.txt) et son contenu () dans un Literal Data packet (Tag 11 présent dans le premier octet) dont le format est

décrit en section 5.9 de la RFC4880. L'octet suivant indique la longueur du paquet : 0x1e, i.e. 30 octets.

```
>>> d = d[2:]
>>> d
'b\x07mdp.txtK\xabd\x0cO7huQcYzHEPSq82m\n'
```

l'octet suivant (`d[1]`) indique le type de données : 'b' pour des données binaires. Il est suivi d'un nom de fichier précédé de sa longueur (mdp.txt' de 7 octets de long), puis d'une date (25 Mars 2010 14 :24 :28) associée au fichier :

```
>>> import datetime
>>> seconds = struct.unpack('!I', d[9:9+4])
>>> datetime.datetime.fromtimestamp(seconds)
datetime.datetime(2010, 3, 25, 14, 24, 28)
```

Ce qui suit correspond au données du fichier, i.e. au mot de passe :

```
>>> print d[13:]
O7huQcYzHEPSq82m

>>>
```

Le mot de passe est donc : **O7huQcYzHEPSq82m**

7.5.5 Conclusion

La forme spécifique (construite pour l'occasion) du modulo associé à la clé ElGamal de Personne a permis de remonter assez facilement à la partie secrète associée et à la clé éphémère utilisée pour le chiffrement du message.

Ceci a ensuite permis d'obtenir la clé de session utilisée pour chiffrer le message et ensuite ce message lui-même.

Cette partie a également été l'occasion de passer un peu de temps sur le format des messages GPG.

Malgré tout, une question (annexe) reste sans réponse : à quoi sert le bloc RSA dans le message chiffré ?

8 Analyse du classes.dex de com.anssi.secret

Pour comprendre précisément les détails du fonctionnement de l'application Secret, il semble judicieux d'analyser son fichier principal, **classes.dex**.

Il n'est pas envisageable d'étudier directement le contenu du .dex. Une étape de décompilation initiale est nécessaire pour convertir celui-ci en primitives de plus haut niveau.

Pour ce faire, on utilise l'outil **baksmali** (équivalent islandais de "désassembleur") du projet smali. On télécharge la dernière version (1.2.2) du jar de l'application et de son script de lancement :

```
$ wget -q http://smali.googlecode.com/files/baksmali-1.2.2.jar
$ wget -q http://smali.googlecode.com/files/baksmali
$ ln -s baksmali-1.2.2.jar baksmali.jar
$ chmod u+x baksmali
```

baksmali est capabale de traiter directement un .apk et de générer un ensemble de fichiers .smali correspondant à l'implémentation des différentes classes présentes dans l'apk :

```
$ ./baksmali com.anssi.secret.apk -o secret/
$ find secret/
secret/
secret/com
secret/com/anssi
secret/com/anssi/secret
secret/com/anssi/secret/R$attr.smali
secret/com/anssi/secret/SecretJNI.smali
secret/com/anssi/secret/R$string.smali
secret/com/anssi/secret/R$layout.smali
secret/com/anssi/secret/R$id.smali
secret/com/anssi/secret/R.smali
secret/com/anssi/secret/RC4.smali
```

Avec un peu de documentation, l'assembleur présent dans les fichiers .smali se lit assez facilement. Une première analyse des 7 fichiers générés met en évidence les 2 principaux : SecretJNI.smali et RC4.smali. Pour le lecteur curieux, ces 2 fichiers sont attachés à ce document. Leur contenu est détaillé ci-dessous.

8.1 SecretJNI.smali

SecretJNI.smali contient les méthodes suivantes :

```
$ grep ^.method SecretJNI.smali
.method static constructor <clinit>()V
.method public constructor <init>()V
.method private dechiffrer(Ljava/lang/String;)[B
.method public static hexStringToByteArray(Ljava/lang/String;)[B
.method public native deriverclef(Ljava/lang/String;[D)Ljava/lang/String;
.method public onClick(Landroid/view/View;)V
.method public onCreate(Landroid/os/Bundle;)V
.method public onLocationChanged(Landroid/location/Location;)V
.method public onProviderDisabled(Ljava/lang/String;)V
.method public onProviderEnabled(Ljava/lang/String;)V
.method public onStatusChanged(Ljava/lang/String;ILandroid/os/Bundle;)V
```

Les méthodes les plus intéressantes sont discutées ci-dessous

8.1.1 clinit()

Le constructeur clinit() contient 3 éléments intéressants :

- Une variable 'coincoin' contenant une chaine qui semble être encodée en base64. Aucune référence à celle-ci n'apparait dans le reste de l'application.

```
>>> coincoin="bmV3c29mdCwgdHUgZXMgaW50ZXJkaXQgZGUgY2hhbGxlbm
... dlIHBvdXIgc29jaWFsIGVuZ2luZWVyaW5nIGV4Y2Vzc2lmLg=="
>>> coincoin.decode('base64')
'newsoft, tu es interdit de challenge pour social engineering excessif.'
```

- une variable 'programme' contenant une chaine de 2548 octets encodées en hexa. Celle-ci est utilisée uniquement dans la méthode dechiffrer() : elle y est convertie en tableau d'octets (via un appel à hexStringToByteArray(), le résultat étant ensuite passé à la méthode crypt() d'une instance de la classe RC4 (initialisée avec une clef passée en paramètre de dechiffrer()). La chaine 'programme' (les retours à la ligne on été rajoutés)

```
"477689b3cb25eba2b9d671cb4a256c07e6bc1902e125970ee14312b2f61976e01a294d2c
80a3edc9a08a800475b31dea9752b68d5b9195af61a0bfcb50870f659ec1ef60329f9b721
ffde227332477392d58b05ba664db3095704b69ffdec2382090a1bf1ec8bca220b1c627b8
a64062ab7fbd7e7c2f6ba61a63dcb02f7ca412ef960a1ae87c8cdd3f026bd8d069426aefc
d9377edfbce82256c6ff9e38f757a82d4e508f93612c062bd3d94feb1fedfc726f307fb02
e00110693a07081872b78fba7f15d80256d784e182793b08519a25edbe2e099cd551fb215
5bc752de734815340f11e2549f597b8d22d483b18d70727f411648363224095d533754208
83cab6191d18464a5a86d2e9172be74af80f0df17b1433445551220f3bea62869516068c7
de3e94bab7a863ce5849a53d15c7da2c0029272a92d7d269c1de47b1ff4a187460b557ebc
72c800d4f367db00985c4135dd2f8d23cade975dafc3b57e44d93b45e9bc0797c9a124e00
c55837c51851f3140a9450d9dd8d18237df92037daf1de8779a9399bc20549d32e9dc7f15
60ab0dfa22f33bb7bc4c4635712afb229175e2da1f400c17f977d2408a447db91decfef8f
767426dc747b67d3b992a8b02ac40290d130cf7289a874e99442c9b64b8b539244ca49661
f190e72f1079f46e96463d7454801876060a24132eb32dcad4c96fecccab472e7617d08f3
3e19b92c6eadf237218d6057db4e0855f3999f09c9f336c1de5bc7e3a1e9fdae589637cc6
c82ddef7c84ea2baf108d44d73b793caa94505f032a5f7d32e38031169c2aa76ba673b223
32bc9b36249c0498024c686550bffed45b8de628b6c1bd062cf00caf88d6e0e8fe9d17418
98c083f4e8b6c4b512e24516c2717cef1ea4bc6b3d96ef572d50286ddcf8e5e969d673e3d
ed48c261b6746838025fa090fb60cf9358e73b94ae09bdd993db5eeb8e232e45cfa20fc34
3f3b2d1392705d0aee69b82f3f2f70d79b354c56ee6ac133b92f4a5930a431ffc7efd2f3f
bc96d7297e6745692fce02e53d908c397e4dab8a681c725add4f1837cfbc6451466edf0b7
47429e93c22ab4d5d0397973df6fcdd5ff34e612b72b83082619f7d6c1f8a152462ffc248
ccc1695c419a74bee4a38ce608af2aa46b5d77cdd7473a7c323a4e295c0a066fcfdf1e67d
ad21d226a899d7f8b5ab054a3bef64f69e2a0d14c1a7e5e7c6bcdbba7d04bf66a4e741a9e
4197350b2a63b245711e97a09b94ef8c1e7942af867b49134b3d24d22d17c3ef8ee835c60
d2b332ce3e4c698795c60779bbda72f2185c3f368c91f0021a843a7abb6a1f3ff7c26b6f8
93e80a983d7126e0fe0ffa18fc628597740ce501009e0bedaed3f4f296f98180b29201f71
6168242cd779b67ab209fcb2bcf89b2e22b0f10cf1876617b947e6e00ac7e9ce696a8f6b3
dfd46986c46dc0d937ab4a05641c76e166ab6222d2a92249c71cb7c16cd04d6ab2eb56ca3
```

```
98fdb57e1079d8be3b5e56eb1f2a474c76ca3ce475461829bb957897ae930cb9e8436423e
cd7d7fd5b2ce481ddbcc84a0b265a5093ba717c5b228e027602367f69dd921d7c8c07b9d6
e73da4960811b2845da888590dc688acb186297b5f06db14b86621790101666f600f46fdb
5653083bfd819b2f1d3e3f61f456c66b7e5737e361b5f3876dd4f58b1cc12aa31018aa7c6
42577094b060164eb3ca799a05f520bd201f03a1bbdfdd8d0605082b7d7f3f4afc3156bf4
9c0c22cc3fd58b3578e845a50caaa034d329324e36cfb09e63f9018f081df0fe3a2"
```

- Un appel à `System->LoadLibrary(‘‘hello-jni’’)`

8.1.2 dechiffrer()

Comme évoqué ci-dessus, la méthode prend un paramètre une chaine “clef”. Elle accède à la variable d'instance ‘programme’ (chaine hexa discutée ci-dessus), qu'elle convertit en byteArray. Le byteArray est ensuite déchiffré (via l'implémentation de RC4) avec la clef passée en paramètre de la méthode. La chaine déchiffrée est retournée.

8.1.3 deriverclef()

L'implémentation de cette méthode virtuelle est fournie par la bibliothèque libhello-jni.so chargée par clinit(). Elle est appelée depuis onClick() et sa sortie est passée à la méthode dechiffrer(). Elle prend en paramètre une référence vers l'environnement JNI, un tableau de lieux (coordonées GPS) et une chaine (mot de passe).

8.1.4 onClick()

Malgré son nom, cette méthode contient finalement le coeur de l'application. L'analyse de son fonctionnement est assez simple. Sa trame est la suivante.

Elle se charge tout d'abord de recueillir les coordonnées GPS (latitude et longitude) des 4 lieux qui sont stockés (après multiplication par π) dans un tableau de double. L'utilisateur se voit ensuite offrir la possibilité d'entrer le mot de passe et de valider.

A ce moment, le tableaux de lieux et le mot de passe sont passés à la méthode virtuelle deriverclef() (dont l'implémentation est fournie par la libhello-jni.so). **A priori**, celle-ci retourne une clé qui est ensuite passée à la méthode dechiffrer() décrite ci-dessus.

En pratique, des coordonnées géographiques invalides passées à l'application font “planter” celle-ci. Les raisons de ce plantage sont détaillés dans la section présentant le reverse de la bibliothèque libhello-jni.so.

La chaine déchiffrée retournée par la fonction est sauvée sur disque dans le répertoire de l'application sous le nom “binaire”. Un message (“Bravo ! Va lancer le binaire pour voir si ça a marché !”) est affiché comme présenté ci-dessous :

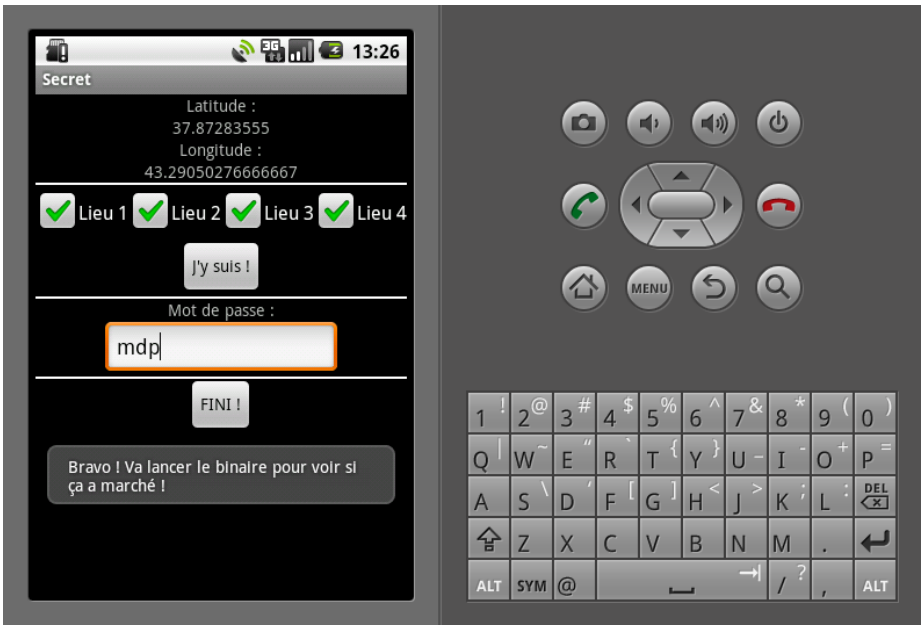

FIGURE 9 – Résultat du passage de bonne coordonnées (peu important le mot de passe)

Le mot de passe influant sur la clé de déchiffrement générée, il est nécessaire que celui-ci soit valide : autrement, le fichier sauvé sur disque est juste inutilisable :

```
$ ./tools/adb shell
# cd /data/data/com.anssi.secret/files
# ls -l
-rw-rw---- app_24   app_24       1274 2010-05-10 13:21 binaire
# chmod 755 binaire
# ./binaire
./binaire: 1: Syntax error: word unexpected (expecting ")")
```

Bien, entendu, le résultat diffère sensiblement avec les bonnes coordonnées et le bon mot de passe :

```
# ./binaire
Bravo, le challenge est terminé! Le mail de validation est : \
4284d974a8af53aa7a85fc4e956b2d84@sstic.org
```

Au final, sauf à avoir trouver les réponses aux 4 questions et à avoir réalisé le déchiffrement du mot de passe chiffré en GPG, les étapes suivantes consistent à

- étudier le traitement des entrées (coordonnées et lieux) par deriverclef(), pour tenter de remonter à la logique de dérivation de clef utilisée, et possiblement à des valeurs valides de ces paramètres d'entrée. Cette compréhension passe par le reverse de la bibliothèque libhello-jni.so, décrit dans la section suivante.
- étudier le module RC4 utilisé pour le déchiffrement. Cette analyse est présentée ci-dessous.

8.2 RC4.smali

Le fichier RC4.smali fournit **a priori** l'implémentation d'un module pour l'algorithme de chiffrement de flux RC4. Il est utilisé directement par SecretJNI.smali mais également par la libhello-jni.so. Les méthodes du module (détaillées ci-dessous) sont les suivantes :

```
$ grep ^.method RC4.smali
.method public constructor <init>([B)V
.method static com([B[B)V
.method public crypt([B)[B
.method public cryptself([B)V
.method getbyte()B
```

Lors de l'étude de la libhello-jni.so, la méthode statique com() est utilisée. Celle-ci prend en paramètre une clé et un tableau de données à chiffrer. L'étude via gdb des entrées (clés et chaine) et sortie (chaine d'entrée chiffrée), et la comparaison de celles-ci avec le résultat obtenu avec une implémentation de RC4 montre que celles-ci diffèrent : soit le module n'implémente pas du tout RC4 comme son nom semble l'indiquer, soit il s'agit d'une implémentation légèrement modifiée de RC4.

Même s'il serait possible de considérer le module de manière opaque et de le réutiliser dans nos calculs, sa taille limitée (300 lignes de smali) permet une étude plus détaillée :

- **public constructor <init>([B)V** : le constructeur init() prend en paramètre une clef passée sous forme de tableau d'octets. Elle est utilisée pour initialiser l'état interne (un tableau de 256 octets) de l'instance RC4.
- **static com([B[B)V** : cette méthode statique de la classe prend en paramètre une clé et des données (toutes deux passées sous forme de tableaux d'octets). La clé permet d'initialiser une instance de l'algorithme de chiffrement et ensuite de chiffrer les données via un appel à cryptself(). Le tableau de données passé en paramètre est modifié en sortie.
- **public crypt([B)[B** : cette méthode est utilisée pour chiffrer des données passées en argument (sous forme d'un tableau d'octets). Elle retourne un tableau d'octets correspondant aux données chiffrées.
- **public cryptself([B)V** : cette méthode est comparable à la méthode crypt() précédente mais chiffre les données passées "sur place" et ne retourne rien.

– **getbyte()B** : Il s'agit d'une méthode interne utilisée par crypt() et crypt-self(). Elle permet d'accéder aux octets de keystream (et réalise donc les modifications associées sur l'état interne de l'algorithme).

Une analyse de la méthode getbyte() avec une implémentation de référence ne montre aucune différence particulière. En revanche, le constructeur init(), même s'il réalise un début d'initialisation dans les règles de l'état interne à partir de la clé, ajoute à celle-ci une étape supplémentaire : il jette les 0xc00 (3072) premiers octets de keystream. A la décharge des développeurs, il semblait déraisonnable de nommer le module RC4DROP3072 ...

Au final, une fonction de chiffrement/déchiffrement équivalente à celle du module est obtenue avec les quelques lignes de Python suivantes :

```
from Crypto.Cipher import ARC4

def dec(key, data):
    return ARC4.new(key).decrypt('Z'*0xc00+data)[0xc00:]
```

9 Reverse de la libhello-jni.so

9.1 Introduction

Comme évoqué précédemment, la dérivation de clé à partir des informations de lieux et du mot de passe est effectuée par la méthode (viruelle) deriverclef(), dont l'implémentation est fournie par la bibliothèque libhello-jni.so.

Cette section détaille le reverse réalisé sur cette bibliothèque pour comprendre comment cette dérivation est effectuée. La finalité de cette opération est également d'obtenir des informations sur les coordonnées et le mot de passe, et éventuellement de remonter (directement ou par brute force) à ceux-ci.

Le reverse de la bibliothèque a été réalisé "à la main" en utilisant uniquement **objdump** et un éditeur de texte (**emacs**). **vi** aurait également fait l'affaire.

9.2 Première passe sur la sortie d'objdump

La libhello-jni.so est obtenue simplement en dézippant l'APK de l'application Secret.

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

Pour désassembler la bibliothèque, on utilise une version d'**objdump** supportant l'architecture ARM. Ce genre d'outil n'est généralement pas disponible directement dans les distributions mais reste assez simple à se procurer :

- Le projet Debian Embedded fournit des versions directement installables de l'outil pour différentes architectures, dont ARM.

- Le NDK Android installé précédemment en fournit une version (build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/arm-eabi-objdump).
- Les possesseurs de machines ARM peuvent installer et utiliser une version native de l'outil. **objdump** est notamment disponible sur Nokia N900.

La dernière solution a été utilisée par l'auteur mais la deuxième reste certainement la plus simple à mettre en oeuvre dans le cas qui nous intéresse :

```
$ cd android-ndk-r3
$ cd build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/
$ ./arm-eabi-objdump -D /tmp/libhello-jni.so  > /tmp/libhello-jni.so.asm
$ ls -lh libhello-jni.so.asm
-rw-r--r-- 1 arno arno 159K May 10 14:50 libhello-jni.so.asm
$ wc -l libhello-jni.so.asm
4101 libhello-jni.so.asm
```

Dans un éditeur texte, on peut donc commencer l'analyse de la bibliothèque. Un premier passage sur son contenu montre que :

- les noms des fonctions de la bibliothèque sont tous obfusqués à l'exception de deriverclef().
- Des parties de la librairie, en mode Thumb (instructions 16 bits), n'ont pas été désassemblées correctement. Ce n'est pas un problème pour le moment. Il est toujours possible de désassembler à nouveau la bibliothèque en utilisant l'option `--disassembler-options=force-thumb` pour obtenir ces parties.
- Des éléments de données ont été désassemblés comme du code, par exemple à la fin de deriverclef() :

```
191a: 46a2       mov sl, r4
191c: 46ab       mov fp, r5
191e: bdf0       pop {r4, r5, r6, r7, pc}
1920: 31b0       adds r1, #176
1922: 0000       lsls r0, r0, #0
1924: eddc ffff  ldcl 15, cr15, [ip, #1020]
1928: edf8 ffff  ldcl 15, cr15, [r8, #1020]!
```

- La fonction deriverclef() compte 250 lignes d'assembleur et fait appel à plusieurs sous-fonctions (aux noms obfusqués : G4Cy, SXXJZ, QUZd7pH7J, ...).

Le début du code de la fonction est présenté ci-dessous :

```
00001728 <Java_com_anssi_secret_SecretJNI_deriverclef>:
    1728: b5f0       push {r4, r5, r6, r7, lr}
    172a: 465f       mov r7, fp
    172c: 4656       mov r6, sl
    172e: 464d       mov r5, r9
    1730: 4644       mov r4, r8
    1732: b4f0       push {r4, r5, r6, r7}
```

```
    1734: b0eb       sub sp, #428
    1736: 9205       str r2, [sp, #20]
    1738: 6802       ldr r2, [r0, #0]
    173a: 4979       ldr r1, [pc, #484] (1920 <Java_..._deriverclef+0x1f8>)
    173c: 4698       mov r8, r3
    173e: 23a9       movs r3, #169
    1740: 009b       lsls r3, r3, #2
    1742: 58d3       ldr r3, [r2, r3]
    1744: 4689       mov r9, r1
    1746: 2200       movs r2, #0
    1748: 9905       ldr r1, [sp, #20]
    174a: 1c07       adds r7, r0, #0
    174c: 4798       blx r3
    174e: 4b75       ldr r3, [pc, #468] (1924 <Java_..._deriverclef+0x1fc>)
    1750: 44f9       add r9, pc
    1752: ae61       add r6, sp, #388
    1754: 444b       add r3, r9
    1756: 9002       str r0, [sp, #8]
    1758: 1c19       adds r1, r3, #0
    175a: 1c32       adds r2, r6, #0
    175c: c931       ldmia r1!, {r0, r4, r5}
    175e: c231       stmia r2!, {r0, r4, r5}
    1760: 6808       ldr r0, [r1, #0]
    1762: ad68       add r5, sp, #416
    1764: 4644       mov r4, r8
    1766: 6010       str r0, [r2, #0]
    1768: 7909       ldrb r1, [r1, #4]
    176a: 7111       strb r1, [r2, #4]
    176c: 695a       ldr r2, [r3, #20]
    176e: 4669       mov r1, sp
    1770: 699b       ldr r3, [r3, #24]
    1772: 3199       adds r1, #153
    1774: 31ff       adds r1, #255
    1776: 604b       str r3, [r1, #4]
    1778: 9103       str r1, [sp, #12]
    177a: 9266       str r2, [sp, #408]
    177c: 2180       movs r1, #128
    177e: aa07       add r2, sp, #28
    1780: 1c10       adds r0, r2, #0
    1782: 0049       lsls r1, r1, #1
    1784: 4693       mov fp, r2
    1786: f7ff fe1f  bl 13c8 <G4Cy>
    178a: 683a       ldr r2, [r7, #0]
...
```

La suite des hostilités consiste simplement à se munir d'une bonne documentation sur l'assembleur ARM [4] et de "lire" le code pour comprendre les opérations réalisées à partir des coordonées GPS et du mot de passe passés en

4. Notamment sur http://infocenter.arm.com/help/index.jsp

paramètre à la fonction.

La suite de cette section fait l'hypothèse pour des raisons de concision que le lecteur est quelque peu familier de l'architecture ARM.

Dans certains cas, il est bon de valider les conclusions associées à la lecture du code en étudiant l'état des registres du processeurs et de la pile de l'application en utilisant pour cela GDB, dont l'installation a été détaillée plus haut.

Présenter ici une explication ligne à ligne de la fonction n'a que peu d'intérêt. La trame de la fonction est donc détaillée ci-après (avec quelques extraits de code) à un niveau un peu plus élévé avec les difficultés rencontrées.

9.3 Trame de deriverclef()

Une première chose à remarquer sur l'application est qu'elle alloue un volume important de place sur sa pile en 0x1734 :

```
1732: b4f0       push {r4, r5, r6, r7}
1734: b0eb       sub sp, #428
1736: 9205       str r2, [sp, #20]
```

Nous verrons que cet espace est utilisé :

- pour des variable locales,
- pour les calculs réalisés par un algorithme de hash,
- pour reconstruire (à partir des coordonnées de lieux et d'élément présent la section rodata de la bibliothèque) le nom d'une classe
- pour la conversion des coordonnées de lieux
- ...

Après une sauvegarde des registres r4 à r12 et l'allocation précédente sur la pile, la fonction réalise un appel (en 0x174c) à la méthode **const char* (*GetStringUTFChars)(JNIEnv*, jstring, jboolean*)** de l'environnement JNI pour convertir le mot de passe en simple chaine de caractères. Les calculs qui précèdent cet appel sont initialement un peu déroutant mais méritent d'être expliqués puisque le mécanisme associé est réutilisé plusieurs fois par la suite :

```
173e: 23a9       movs r3, #169
1740: 009b       lsls r3, r3, #2
1742: 58d3       ldr r3, [r2, r3]
```

Le code précédent calcule une valeur de décalage fixe ($169 \ll 2 = 169*4$) dans la structure JNINativeInterface (vers laquelle r2 pointe). Le fichier définissant la structure est présent dans le NDK Android :

```
$ cd android-ndk-r3/
$ less build/platforms/android-5/arch-arm/usr/include/jni.h
...
typedef const struct JNINativeInterface* JNIEnv;
```

```
...

struct JNINativeInterface {
    void*       reserved0;
    void*       reserved1;
    void*       reserved2;
    void*       reserved3;

    jint        (*GetVersion)(JNIEnv *);

    jclass      (*DefineClass)(JNIEnv*, const char*, jobject, const jbyte*,
                        jsize);
    jclass      (*FindClass)(JNIEnv*, const char*);

    jmethodID   (*FromReflectedMethod)(JNIEnv*, jobject);
    jfieldID    (*FromReflectedField)(JNIEnv*, jobject);
    /* spec doesn't show jboolean parameter */
    jobject     (*ToReflectedMethod)(JNIEnv*, jclass, jmethodID, jboolean);

    jclass      (*GetSuperclass)(JNIEnv*, jclass);
    jboolean    (*IsAssignableFrom)(JNIEnv*, jclass, jclass);

    /* spec doesn't show jboolean parameter */
    jobject     (*ToReflectedField)(JNIEnv*, jclass, jfieldID, jboolean);

    jint        (*Throw)(JNIEnv*, jthrowable);
...
```

Au final, ce calcul permet d'accéder au $169^{ème}$ élément de la structure, i.e.

```
const char* (*GetStringUTFChars)(JNIEnv*, jstring, jboolean*);
```

Après cet appel, sont copiés sur la pile (à partir de sp+388) 17 octets provenant de la section rodata. Ceux-ci seront utilisés plus tard en association avec les coordonnées de lieux pour construire un nom de classe. 3 octets après la fin des données copiés (en sp+408) sont à nouveau copiés 8 octets de la section rodata : ceux-ci serviront plus tard comme signature pour l'appel à une méthode statique de la classe évoquée précédemment.

Utilisont GDB pour obtenir le contenu de la pile à sp+388 après la copie en plaçant un breakpoint, par exemple en 0x1780. Une fois l'application lancée dans l'émulateur, **ps** permet d'obtenir son PID : 298 sur notre exemple. En étudiant le contenu de cat **/proc/298/maps**, on obtient l'adresse à laquelle la bibliothèque a été chargée (fixe d'un lancement de l'application sur l'autre).

```
80a00000-80a04000 r-xp 00000000 1f:01 505 /data/ ... /lib/libhello-jni.so
80a04000-80a05000 rwxp 00003000 1f:01 505 /data/ ... /lib/libhello-jni.so
```

En dehors de l'émulateur :

```
$ ./build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/arm-eabi-gdb
...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0xafe0da04 in ?? ()
(gdb) b *0x80a01780
Breakpoint 1 at 0x80a01780
(gdb) b *0x80a01781                    # nécessaire pour une
Breakpoint 2 at 0x80a01781             # raison inconnue
(gdb) c
Continuing.

Breakpoint 1, 0x80a01780 in ?? ()
(gdb) i r
...
sp             0xbed55748       0xbed55748
...
(gdb) x/28b 0xbed55748+388
0xbed558cc:   0xf4  0x94  0x7d  0x75  0x17  0xee  0x04  0xfb
0xbed558d4:   0xfe  0xd4  0x63  0x3f  0x15  0xf2  0x12  0xfc
0xbed558dc:   0xb8  0x01  0x00  0x00  0xd7  0xa4  0xbd  0xa4
0xbed558e4:   0xbd  0xd6  0xa9  0xff
```

Nous retrouvons bien les octets copiés de la section rodata. Nous reviendrons sur leur utilisation un peu plus tard.

Ensuite, le code de deriverclef() fait appel à une fonction de la bibliothèque au nom obfusqué <G4Cy> : les paramètres passées (via r0 et r1) à celle-ci sont un pointeur vers un tampon statique dans la pile (en sp+28) et la valeur 256. Une analyse rapide du code de la fonction et quelques indices supplémentaires obtenus en continuant la lecture de deriverclef() laissent à penser qu'il s'agit d'une fonction d'initialisation d'un algorithme de hash :

```
000013c8 <G4Cy>:
    13c8: b5f0    push {r4, r5, r6, r7, lr}
    13ca: 465f    mov r7, fp
    13cc: 4656    mov r6, sl
    13ce: 464d    mov r5, r9
    13d0: 4644    mov r4, r8
    13d2: b4f0    push {r4, r5, r6, r7}
    13d4: 2380    movs r3, #128            # r3 = 128
    13d6: 468a    mov sl, r1               # sl = r1
    13d8: 005b    lsls r3, r3, #1          # r3 = 128<<1 = 256
    13da: b093    sub sp, #76
    13dc: 1c06    adds r6, r0, #0
    13de: 459a    cmp sl, r3               # comparaison avec 256
    13e0: d00f    beq.n 1402 <G4Cy+0x3a>
    13e2: 459a    cmp sl, r3
    13e4: dd09    ble.n 13fa <G4Cy+0x32>
    13e6: 23c0    movs r3, #192            # r3 = 192
    13e8: 005b    lsls r3, r3, #1          # r3 = 192<<1 = 384
```

```
13ea: 459a    cmp sl, r3              # comparaison avec 384
13ec: d009    beq.n 1402 <G4Cy+0x3a>
13ee: 2380    movs r3, #128            # r3 = 128
13f0: 009b    lsls r3, r3, #2          # r3 = 128<<2 = 512
13f2: 459a    cmp sl, r3              # comparaison avec 512
13f4: d005    beq.n 1402 <G4Cy+0x3a>
13f6: 2002    movs r0, #2
13f8: e06d    b.n 14d6 <G4Cy+0x10e>
13fa: 29c0    cmp r1, #192            # comparaison avec 192
13fc: d001    beq.n 1402 <G4Cy+0x3a>
13fe: 29e0    cmp r1, #224            # comparaison avec 224
1400: d1f9    bne.n 13f6 <G4Cy+0x2e>
1402: 4651    mov r1, sl
```

En voyant les valeurs avec lesquelles le second paramètre est comparé on pense **initialement** à la série des SHA2. Cette hypothèse sera infirmée par la suite.

Dans la suite du code, un jbytearray de 32 éléments est alloué (en 0x1796) via un appel à la $176^{ème}$ fonction du JNIEnv :

```
jbyteArray    (*NewByteArray)(JNIEnv*, jsize);
```

Celui-ci sera utilisé plus tard pour stocker la sortie de l'algorithme de hash appelé sur le mot de passe de manière à pouvoir la passer à une fonction du module RC4.

Une fois l'allocation effectuée, le code de deriverclef() réalise (entre 0179c et 17a6) un NOT de chacun des 8 octets à partir de sp+408. Il s'agit des 8 derniers octets obtenus précédemment avec gdb. Une vérification de la même zone mémoire après cette boucle le confirme :

```
(gdb) x/28b 0xbed55748+388
0xbed558cc:   0xf4  0x94  0x7d  0x75  0x17  0xee  0x04  0xfb
0xbed558d4:   0xfe  0xd4  0x63  0x3f  0x15  0xf2  0x12  0xfc
0xbed558dc:   0xb8  0x01  0x00  0x00  0x28  0x5b  0x42  0x5b
0xbed558e4:   0x42  0x29  0x56  0x00
```

La suite de deriverclef() devient plus intéressante, car elle traite les coordonnées de lieux passées en paramètre de la fonction. Un appel à la $190^{ème}$ fonction du JNIEnv

```
jdouble* GetDoubleArrayElements(JNIEnv*, jdoubleArray, jboolean*)
```

permet d'accéder un par un aux éléments du jdoubleArray contenant les coordonnées GPS sous forme de jdouble (i.e. double, i.e. 64-bit IEEE 754). Chacun d'entre eux est ensuite passé successivement à 2 fonctions inconnues présentes dans la bibliothèque :

```
17d0: f000 e8ac  blx 192c <SXXJZ>
17d4: f000 ed2c  blx 2230 <QUZd7pH7J>
```

Une analyse du code de ces 2 fonctions montre un grand nombre de calculs mathématiques sur les entrées et des appels à quelques sous-fonctions également présentes dans le module. Avant de se lancer dans une analyse détaillée "à la main" de la fonction, il semble judicieux de vérifier que celles-ci ne sont pas simplement des fonctions de bibliothèques classiques (libc, libm, ...) intégrées statiquement à la libhello-jni.so. Pour cela, il suffit de désassembler les bibliothèques classiques avec **objdump** et de chercher un motif spécifique. Par exemple, le `pop {r4, r5, r6, r7, pc}` en plein centre de la première fonction est un bon candidat :

```
 19ac: e28dd004  add sp, sp, #4 ; 0x4
 19b0: e8bd80f0  pop {r4, r5, r6, r7, pc}
 19b4: e3a045ff  mov r4, #1069547520 ; 0x3fc00000
```

On le retrouve rapidement dans le code désassemblé de la libm, en plein centre de la fonction round() (la libm présente sur le système n'est évidemment pas obfusquée). Une analyse rapide montre que le code des 2 fonctions est identique :

```
0001b09c <round>:
   1b09c: e92d40f0  push {r4, r5, r6, r7, lr}
   1b0a0: e24dd004  sub sp, sp, #4 ; 0x4
   1b0a4: e1a04000  mov r4, r0
   1b0a8: e1a05001  mov r5, r1
   1b0ac: ebfff83e  bl 191ac <__isfinite>
   1b0b0: e3500000  cmp r0, #0 ; 0x0
   1b0b4: 0a000016  beq 1b114 <round+0x78>
   1b0b8: e1a00004  mov r0, r4
   1b0bc: e1a01005  mov r1, r5
   1b0c0: e3a02000  mov r2, #0 ; 0x0
   1b0c4: e3a03000  mov r3, #0 ; 0x0
   1b0c8: ebff9c07  bl 20ec <__isinf-0x48>
   1b0cc: e3500000  cmp r0, #0 ; 0x0
   1b0d0: 0a00001c  beq 1b148 <round+0xac>
   1b0d4: e1a00004  mov r0, r4
   1b0d8: e1a01005  mov r1, r5
   1b0dc: ebfff3b1  bl 17fa8 <floor>
   1b0e0: e1a02004  mov r2, r4
   1b0e4: e1a03005  mov r3, r5
   1b0e8: e1a06000  mov r6, r0
   1b0ec: e1a07001  mov r7, r1
   1b0f0: ebff9c00  bl 20f8 <__isinf-0x3c>
   1b0f4: e3a034bf  mov r3, #-1090519040 ; 0xbf000000
   1b0f8: e3a02000  mov r2, #0 ; 0x0
   1b0fc: e283360e  add r3, r3, #14680064 ; 0xe00000
   1b100: ebff9be1  bl 208c <__isinf-0xa8>
   1b104: e3500000  cmp r0, #0 ; 0x0
   1b108: 1a000005  bne 1b124 <round+0x88>
```

```
   1b10c: e1a04006  mov r4, r6
   1b110: e1a05007  mov r5, r7
   1b114: e1a00004  mov r0, r4
   1b118: e1a01005  mov r1, r5
   1b11c: e28dd004  add sp, sp, #4 ; 0x4
   1b120: e8bd80f0  pop {r4, r5, r6, r7, pc}
   1b124: e3a045ff  mov r4, #1069547520 ; 0x3fc00000
...
```

La seconde fonction (QUZd7pH7J) est une fonction de conversion de double en entier (son code est quasiment identique à celui de `__aeabi_d2iz()`) présente dans la libc.

Au final, la boucle entre 0x17c2 et 0x17de se charge transformer (via arrondi et conversion en entier) le tableau de coordonnées GPS en un simple tableau de 8 octets. Pour chaque valeur de coordonnée passée par l'utilisateur, le résultat dans le tableau de 8 octets est la sortie de l'équivalent Python suivant :

```
def convert_coord(x):
    return int(round(x*3.14159265)) & 0xff
```

Le tableau de 8 octets est stocké en pile à partir de sp+416.

Ensuite, deriverclef() réalise un appel à strlen() [5] pour connaitre la longeur du mot de passe.

Les éléments suivants sont ensuite passés à la fonction **8j3zIX** :

- un pointeur vers la zone de donnée initialisée en pile en sp+28 [6].
- un pointeur vers le mot de passe
- la longueur du mot de passe retournée par **strlen()**
- 0

Celle-ci est @ nouveau appelée, cette fois-ci avec les paramètres suivants :

- le même pointeur vers la zone de donnée initialisée en pile en sp+28
- un pointeur vers le tableau de 8 octets de coordonnées converties
- 32
- 0

Un point important ici concerne le $3^{ème}$ paramètre. Lors du premier appel à la fonction, le mot de passe est passé avec sa taille en nombre d'octets. Pour le tableau de 8 octets, la valeur 32 est passée. Une analyse du début de la fonction **8j3zIX** laisse à penser que celle-ci considère son troisième paramètre comme

5. l'appel se fait via la plt et la got et doit donc être résolu à la main
6. par la fonction que nous soupçonnons être une fonction d'initialisation d'un algorithme de hash

la longueur de son second paramètre en nombre de bits. Si nos hypothèses se confirment, il pourrait s'agir d'une fonction d'update de l'algorithme de hash.

Après ces 2 appels, deriverclef() réalise un appel à **sdlHj** avec pour paramètres la zone en pile en sp+28 et un pointeur vers sp+356. Le second paramètre semble être un buffer de sortie statique :

- Des données en sp+388 laisse à penser à un buffer de 32 octets.
- La fonction suivante appelée par deriverclef() est SetByteArrayRegion() du JNIEnv qui se charge de copier les 32 octets en sp+356 vers le jbytearray alloué au début de la fonction.

Si nos hypothèses se confirment, il pourrait donc bien s'agir de la fonction de finalisation de l'algorithme de hash. La sortie sur 32 octets (i.e. 256 bits) correspond à la valeur passé (en bits) à la fonction d'initialisation. Le reste du feuilleton sur l'algorithme de hash supposé continue dans la sous-section suivante.

La suite de deriverclef() (en 0x1828) réalise un XOR des 17 octets précédemment copiés depuis la section rodata de la bibliothèque (et ensuite NOTés) en sp+288 avec les 8 octets de coordonnés de lieux (répétés). Le résultat est placé progressivement en sp+288. Les quelques lignes suivantes de la fonction finissent le travail en prenant les 4 premiers octets placés en sp+288 et en les XORant avec 3 valeurs fixes présentes dans la fonctions (49, 44, 89, 47) pour les ajouté ensuite après les 17 premiers :

```
1848: 7822   ldrb r2, [r4, #0]  // r4 pointe en sp+288
184a: 2331   movs r3, #49
184c: 1c38   adds r0, r7, #0
184e: 4053   eors r3, r2
1850: 7862   ldrb r2, [r4, #1]
1852: 7463   strb r3, [r4, #17]
1854: 232c   movs r3, #44
1856: 4053   eors r3, r2
1858: 78a2   ldrb r2, [r4, #2]
185a: 74a3   strb r3, [r4, #18]
185c: 2359   movs r3, #89
185e: 4053   eors r3, r2
1860: 78e2   ldrb r2, [r4, #3]
1862: 74e3   strb r3, [r4, #19]
1864: 232f   movs r3, #47
1866: 4053   eors r3, r2
1868: 7523   strb r3, [r4, #20]
```

Si les coordonnées GPS rentrées sont invalides, la chaine résultat est invalides. Les 2 première questions étant offertes, on obtient initialement une chaine partiellement valide comme la suivante :

```
$ cd android-ndk-r3
```

```
$ cd build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/
$ ./arm-eabi-gdb
...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0xafe0da04 in ?? ()
(gdb) b *0x80a0186a
Breakpoint 1 at 0x80a0186a
(gdb) b *0x80a0186b
Breakpoint 2 at 0x80a0186b
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x80a01868 in ?? ()
(gdb) x/1s 0xbed55748+288
0xbed55868:      "com/\a.\024.i/se\005.\002./RC4"
```

Cette chaine étant passée juste après à la méthode FindClass() du JNIEnv, il est nécessaire de la rendre valide en trouvant des coordonnées GPS adaptées. L'implémentation de RC4 faisant partie de l'application, la chaine attendue semble assez évidente : **com/anssi/secret/RC4**. Trouver les 4 derniers octets du tableaux de lieux puis remonter aux valeurs à passer à l'application est immédiat.

Le jeu de valeurs valides (lat, long) est :

- 48.07, 79.90
- 5.10, 28.65
- 37.57, 40.75
- 37.88, 43.30

On confirme via gdb que le nom de classe est correct dans la pile avec ces valeurs :

```
$ cd android-ndk-r3
$ cd build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/
$ ./arm-eabi-gdb
...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0xafe0da04 in ?? ()
(gdb) b *0x80a0186a
Breakpoint 1 at 0x80a0186a
(gdb) b *0x80a0186b
Breakpoint 2 at 0x80a0186b
(gdb) c
```

```
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x80a01868 in ?? ()
(gdb) x/1s 0xbed55748+288
0xbed55868:      "com/anssi/secret/RC4"
```

La suite de deriverclef() est assez simple. Après l'appel à FindClass(), un appel à GetStaticMethod() pour accéder à une méthode statique est réalisé, suivi d'un appel à CallStaticVoidMethod() pour appeler cette méthode. La seule méthode static du module RC4 est la méthode com(). La sortie de l'algorithme de hash supposé (32 octets précédemment placés dans un jbytearray) est passé comme clé mais également comme donnée. Le résultat obtenu va ensuite être formaté (sous forme de chaine hexa) pour être retourné par deriverclef() à l'appelant.

La sous-section suivante discute de la fonction de hachage supposée. La sous-section qui la suit termine la récupération du mot de passe sur la base des informations obtenues dans la sous-section courante.

9.4 La fonction de hash supposée

Pour confirmer nos hypothèses sur l'utilisation d'une fonction de hash dans deriverclef(), nous commençons par analyser ce qui se trouve en pile dans les octets entre sp+28 et sp+288. **gdb** est notre ami :

```
$ cd android-ndk-r3
$ cd build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/
$ ./arm-eabi-gdb
...
(gdb)  target remote localhost:1234
Remote debugging using localhost:1234
0xafe0da04 in ?? ()
(gdb) b *0x80a0178a
Breakpoint 1 at 0x80a0178a
(gdb) b *0x80a0178b
Breakpoint 2 at 0x80a0178b
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x80a01788 in ?? ()
(gdb) i r
...
sp             0xbed55748       0xbed55748
...
(gdb) x/256b 0xbed55748+28
```

```
0xbed55764:   0x08  0x8d  0x03  0x41  0x74  0xbb  0xe3  0xaf
0xbed5576c:   0xb0  0xf3  0xe0  0xaf  0x00  0x00  0x00  0x00
0xbed55774:   0xc0  0xf2  0xe0  0xaf  0xbc  0xb9  0xe3  0xaf
0xbed5577c:   0xb8  0x01  0x00  0x00  0xb4  0x2b  0x00  0x00
0xbed55784:   0xdc  0x00  0x00  0x00  0x80  0xf3  0x00  0xad
0xbed5578c:   0x9b  0xb3  0xe0  0xaf  0x80  0xf3  0x00  0xad
0xbed55794:   0xf9  0x7c  0x05  0xad  0x00  0x00  0x00  0x00
0xbed5579c:   0x98  0x0d  0x12  0x00  0xc0  0x07  0xd2  0x43
0xbed557a4:   0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0xbed557ac:   0x00  0x01  0x00  0x00  0x52  0x45  0xf8  0x52
0xbed557b4:   0x99  0x79  0x4b  0xe5  0xec  0xe3  0x8e  0x2d
0xbed557bc:   0x91  0x51  0x64  0xb9  0x86  0x8b  0x07  0xe0
0xbed557c4:   0xc9  0x44  0x7c  0xbb  0xca  0xc1  0xb5  0xd2
0xbed557cc:   0x8c  0xeb  0xd2  0xb0  0x45  0x5a  0xce  0x14
0xbed557d4:   0xdc  0x50  0xaf  0x22  0x6b  0xbc  0xfd  0xef
0xbed557dc:   0x4a  0xb7  0x21  0xeb  0xee  0xc6  0x55  0xb5
0xbed557e4:   0x96  0x05  0x71  0x3e  0x2f  0x65  0x2a  0xa7
0xbed557ec:   0x5f  0x51  0x01  0x93  0xfa  0xc1  0x28  0xda
0xbed557f4:   0x68  0xd8  0x6f  0x69  0x72  0xbf  0xb6  0x9c
0xbed557fc:   0x02  0x40  0xfe  0x0a  0x15  0x36  0xe0  0xa6
0xbed55804:   0xd4  0xc1  0x38  0x51  0x06  0x63  0x21  0xbe
0xbed5580c:   0x90  0x88  0x8b  0xb3  0x6b  0xb9  0xa8  0x3e
0xbed55814:   0xe4  0xac  0x99  0x32  0xd4  0x4d  0x92  0x30
0xbed5581c:   0xa5  0x34  0xcb  0x55  0x31  0xf0  0x05  0xb4
0xbed55824:   0xba  0x3e  0x23  0xc4  0x79  0x39  0x73  0xb3
0xbed5582c:   0x55  0x9d  0xdd  0xc0  0xae  0x28  0x1c  0xc5
0xbed55834:   0xe1  0xb8  0x27  0xa3  0x67  0x61  0xc5  0x56
0xbed5583c:   0x33  0x44  0x61  0xed  0x60  0x9d  0xb5  0x88
0xbed55844:   0xba  0xce  0xe2  0x60  0x8b  0x4b  0x8b  0x75
0xbed5584c:   0x7f  0x2a  0xe8  0x83  0x28  0x88  0x96  0xbc
0xbed55854:   0xf7  0x0b  0xe0  0xe6  0x55  0x9e  0x83  0xba
0xbed5585c:   0x60  0x1c  0x49  0x9b  0x00  0x00  0x00  0x00
```

Plutôt que de comparer les valeurs du tableau avec des valeurs d'initialisation de fonctions de hash existantes pour différentes tailles de hash, une méthode plus simple est préférée : laisser Google faire le travail.

Une recherche des 8 octets (``b405f031 55cb34a5'') au milieu du buffer (à partir de 0xbed5581c ci-dessus) donne uniquement 2 réponses, pour le même algorithme de hash : Shabal.

Shabal est une fonction de hash soumise par le projet de recherche SAPHIR à la compétition international du NIST sur les fonctions de hash afin de trouver une nouvelle fonction de hachage (SHA-3). La DCSSI (ancien nom de l'ANSSI) fait partie des contributeurs et la fonction supporte des tailles de condensat de 192, 224, 256, 384 and 512 bits.

Le document de soumission de Shabal (un PDF de 300 pages) contient en Appendice A une implémentation de référence. Les fonctions d'init et d'update

prennent des longueurs en nombre de bits.

Une extraction de cette implémentation de référence et l'ajout d'un main() permettent de réaliser quelques tests pour vérifier s'il sagit bien de Shabal ou d'une version modifiée.

Le code résultant est attaché à ce document (shabal.c, shabal.h). Le main() est donné ci-dessous puis commenté :

```
int main(int argc, char *argv[]) {
        hashState state;
        BitSequence buf[32];
        BitSequence geo[8];
        BitSequence mdp[5];
        int i, b0, b1, b2, b3, len;

        geo[0] = 0x97; //
        geo[1] = 0xfb; // geo fix 79.90 48.07
        geo[2] = 0x10; //
        geo[3] = 0x5a; // geo fix 28.65 5.10
        geo[4] = 0x76; //
        geo[5] = 0x80; // geo fix 40.75 37.57
        geo[6] = 0x77; //
        geo[7] = 0x88; // geo fix 43.30 37.88

        mdp[0] = atoi(argv[1])&0xff;
        mdp[1] = atoi(argv[2])&0xff;
        mdp[2] = atoi(argv[3])&0xff;
        mdp[3] = atoi(argv[4])&0xff;
        mdp[4] = 0;

        len = atoi(argv[5]);
        memset(&state, 0, sizeof(hashState));
        Init(&state, 256);
        Update(&state, mdp, len);
        Update(&state, geo, 32);
        Final(&state, buf);
        for(i=0; i<32; i++)
                printf("%02x", buf[i]);
        printf("\n");
}
```

Les coordonnées GPS y sont fixées en dur (valides). Les 4 premiers octets du mot de passe (les valeurs décimales des caractères associés [7]) peuvent être passés en argument. Le $5^{ème}$ argument accepté par le programme est la longueur du mot de passe en nombre de bits, passé directement à la fonction d'update. Aucune vérification n'est effectuée sur les paramètres d'entrées.

7. pour simplifier l'instrumentation ultérieure du programme

Cette implémentation permet de vérifier l'hypothèse émise précédemment selon laquelle la taille du mot de passe en octets est passée alors que la fonction d'update attend une valeur en nombre de bits.

```
$ gcc shabal.c -o shabal
$ ./shabal 116 111 116 111 4     // "toto"
78e765d316db7e2377be6f3598b38089edc2842ab408c39bf6795e6a4f8cd7a0
$ ./shabal 116 111 116 111 8
0ba0d3aa53fbe98dd75c06f3ed2e608b5ffc05b276ac82fb04379b3116eef004
$ ./shabal 116 111 116 111 9
0ba0d3aa53fbe98dd75c06f3ed2e608b5ffc05b276ac82fb04379b3116eef004
$ ./shabal 116 111 116 111 16
be586f0180baed43c2944588b4e7bf56ec983ee8b3c85b078607d46ae96a6fd0
```

Les tests s'effectuent en passant les coordonnées GPS présentes dans le source et 4 mots de passe commençant par "toto" à Secret dans l'émulateur : "toto", "totototo", "tototot" et "totototototototo".

Pour "toto", on obtient le résultat ci-dessous :

```
$ cd android-ndk-r3
$ cd build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/
$ ./arm-eabi-gdb
...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0xafe0da04 in ?? ()
(gdb) b *0x80a01824          // Avant l'export vers le jbytearray
Breakpoint 1 at 0x80a01824
(gdb) b *0x80a01825
Breakpoint 2 at 0x80a01825
(gdb) c
Continuing.

Breakpoint 1, 0x80a01824 in ?? ()
(gdb) i r
...
sp             0xbed55748       0xbed55748
...
(gdb) x/32b 0xbed55748+356
0xbed558ac:   0x78  0xe7  0x65  0xd3  0x16  0xdb  0x7e  0x23
0xbed558b4:   0x77  0xbe  0x6f  0x35  0x98  0xb3  0x80  0x89
0xbed558bc:   0xed  0xc2  0x84  0x2a  0xb4  0x08  0xc3  0x9b
0xbed558c4:   0xf6  0x79  0x5e  0x6a  0x4f  0x8c  0xd7  0xa0
```

On retrouve le résultat obtenu avec notre premier appel à notre programme de test ci-dessus (`./shabal 116 111 116 111 4` avait donné `78e765d31...`).

On vérifie de la même manière pour les 3 autres. Respectivement, pour “toto-toto”, “tototototot” et “totototototototo” passés à l'application, on obtient :

```
(gdb) x/32b 0xbed55748+356
0xbed558ac:   0x0b  0xa0  0xd3  0xaa  0x53  0xfb  0xe9  0x8d
0xbed558b4:   0xd7  0x5c  0x06  0xf3  0xed  0x2e  0x60  0x8b
0xbed558bc:   0x5f  0xfc  0x05  0xb2  0x76  0xac  0x82  0xfb
0xbed558c4:   0x04  0x37  0x9b  0x31  0x16  0xee  0xf0  0x04

(gdb) x/32b 0xbed55748+356
0xbed558ac:   0x0b  0xa0  0xd3  0xaa  0x53  0xfb  0xe9  0x8d
0xbed558b4:   0xd7  0x5c  0x06  0xf3  0xed  0x2e  0x60  0x8b
0xbed558bc:   0x5f  0xfc  0x05  0xb2  0x76  0xac  0x82  0xfb
0xbed558c4:   0x04  0x37  0x9b  0x31  0x16  0xee  0xf0  0x04

(gdb) x/32b 0xbed55748+356
0xbed558ac:   0xbe  0x58  0x6f  0x01  0x80  0xba  0xed  0x43
0xbed558b4:   0xc2  0x94  0x45  0x88  0xb4  0xe7  0xbf  0x56
0xbed558bc:   0xec  0x98  0x3e  0xe8  0xb3  0xc8  0x5b  0x07
0xbed558c4:   0x86  0x07  0xd4  0x6a  0xe9  0x6a  0x6f  0xd0
```

Nos tests confirment donc toutes les hypothèses :

- L'algorithme de hash utilisé est bien Shabal (256)
- La taille passée à la fonction d'update pour le mot de passe est la longueur de celui-ci en octet alors que celle-ci s'attend à recevoir une longueur en nombre de bit.

La conclusion est donc que l'entropie apportée par le mot de passe à la fonction de dérivation est extrêmement réduite du fait du point précédent : 8 bits pour un mot de passe de 1 à 7 caractères, 16 bits pour un mot de passe de 8 à 15 caractères, ...

Nous utilisons ceci dans la section suivante pour réaliser un brute force de manière à retrouver un mot de passe valide et le binaire.

10 Obtention d'un mot de passe et du binaire

Avant de discuter de la récupération du mot de passe attendu par Secret et donc du binaire, il est bon de refaire un point sur les résultats obtenus dans la section précédente :

- Le binaire chiffré est disponible (en dur dans SecretJNI.smali)
- L'algorithme de chiffrement utilisé RC4DROP3072
- La clé est obtenue par un appel à deriverclef()
- La clé retournée par deriverclef() est un hash chiffré en RC4DROP3072. La clé est le hash lui-même.

- L'algorithme de hash utilisé par la fonction de dérivation de clé est Shabal (256)
- Les données hachées sont les 8 octets de lieux et les N premiers caractères du mot de passe ($N = len(mdp)/8$)
- Les 8 octets de lieux ont déjà été récupérés précédemment

Il reste donc simplement à laisser de côté l'émulateur pour réaliser un brute force sur le mot de passe et tester si la clé dérivée permet de déchiffrer un binaire valide.

Ceci s'écrit en quelques lignes de Python en faisant de simples appels au programme **shabal** précédent. Ce n'est pas le plus efficace mais l'espace à couvrir est extrêmement faible ($1 + 256 + 256^2 < 66000$) pour tester tous les mots de passe de 23 caractères ou moins. Le code suivant teste les mots de passe de 16 à 23 caractères.

La condition d'arrêt (hypothèse qui s'est avérée valide) est que le binaire attendu est au format ELF. Le programme **shabal** doit être présent dans le répertoire courant.

```
import popen2
from Crypto.Cipher import ARC4

encbin="""477689b3cb25eba2b9d671cb4a256c07e6bc1902e125970ee14
312b2f61976e01a294d2c80a3edc9a08a800475b31dea9752b68d5b9195af
61a0bfcb50870f659ec1ef60329f9b721ffde227332477392d58b05ba664d
b3095704b69ffdec2382090a1bf1ec8bca220b1c627b8a64062ab7fbd7e7c
2f6ba61a63dcb02f7ca412ef960a1ae87c8cdd3f026bd8d069426aefcd937
7edfbce82256c6ff9e38f757a82d4e508f93612c062bd3d94feb1fedfc726
f307fb02e00110693a07081872b78fba7f15d80256d784e182793b08519a2
5edbe2e099cd551fb2155bc752de734815340f11e2549f597b8d22d483b18
d70727f411648363224095d53375420883cab6191d18464a5a86d2e9172be
74af80f0df17b1433445551220f3bea62869516068c7de3e94bab7a863ce5
849a53d15c7da2c0029272a92d7d269c1de47b1ff4a187460b557ebc72c80
0d4f367db00985c4135dd2f8d23cade975dafc3b57e44d93b45e9bc0797c9
a124e00c55837c51851f3140a9450d9dd8d18237df92037daf1de8779a939
9bc20549d32e9dc7f1560ab0dfa22f33bb7bc4c4635712afb229175e2da1f
400c17f977d2408a447db91decfef8f767426dc747b67d3b992a8b02ac402
90d130cf7289a874e99442c9b64b8b539244ca49661f190e72f1079f46e96
463d7454801876060a24132eb32dcad4c96fecccab472e7617d08f33e19b9
2c6eadf237218d6057db4e0855f3999f09c9f336c1de5bc7e3a1e9fdae589
637cc6c82ddef7c84ea2baf108d44d73b793caa94505f032a5f7d32e38031
169c2aa76ba673b22332bc9b36249c0498024c686550bffed45b8de628b6c
1bd062cf00caf88d6e0e8fe9d1741898c083f4e8b6c4b512e24516c2717ce
f1ea4bc6b3d96ef572d50286ddcf8e5e969d673e3ded48c261b6746838025
fa090fb60cf9358e73b94ae09bdd993db5eeb8e232e45cfa20fc343f3b2d1
392705d0aee69b82f3f2f70d79b354c56ee6ac133b92f4a5930a431ffc7ef
d2f3fbc96d7297e6745692fce02e53d908c397e4dab8a681c725add4f1837
cfbc6451466edf0b747429e93c22ab4d5d0397973df6fcdd5ff34e612b72b
83082619f7d6c1f8a152462ffc248ccc1695c419a74bee4a38ce608af2aa4
6b5d77cdd7473a7c323a4e295c0a066fcfdf1e67dad21d226a899d7f8b5ab
```

```
054a3bef64f69e2a0d14c1a7e5e7c6bcdbba7d04bf66a4e741a9e4197350b
2a63b245711e97a09b94ef8c1e7942af867b49134b3d24d22d17c3ef8ee83
5c60d2b332ce3e4c698795c60779bbda72f2185c3f368c91f0021a843a7ab
b6a1f3ff7c26b6f893e80a983d7126e0fe0ffa18fc628597740ce501009e0
bedaed3f4f296f98180b29201f716168242cd779b67ab209fcb2bcf89b2e2
2b0f10cf1876617b947e6e00ac7e9ce696a8f6b3dfd46986c46dc0d937ab4
a05641c76e166ab6222d2a92249c71cb7c16cd04d6ab2eb56ca398fdb57e1
079d8be3b5e56eb1f2a474c76ca3ce475461829bb957897ae930cb9e84364
23ecd7d7fd5b2ce481ddbcc84a0b265a5093ba717c5b228e027602367f69d
d921d7c8c07b9d6e73da4960811b2845da888590dc688acb186297b5f06db
14b86621790101666f600f46fdb5653083bfd819b2f1d3e3f61f456c66b7e
5737e361b5f3876dd4f58b1cc12aa31018aa7c642577094b060164eb3ca79
9a05f520bd201f03a1bbdfdd8d0605082b7d7f3f4afc3156bf49c0c22cc3f
d58b3578e845a50caaa034d329324e36cfb09e63f9018f081df0fe3a2"""
bin=encbin.replace('\n', '').decode('hex')

def dec(key, data):
    return ARC4.new(key).decrypt('Z'*0xc00+data)[0xc00:]

def bruteforce():
    for j in range(256):
        for i in range(256):
            sout, sin = popen2.popen2("./shabal %d %d 0 0 %d" % (i,j, 23))
            k = sout.read()[:64]
            sout.close()
            sin.close()
            k = k.decode('hex')
            k = dec(k,k)
            k = k.encode('hex')
            d = dec(k, bin)
            if d[1:4] == 'ELF':
                print "[+] Binaire trouvé! "
                print "[+] Début du mot de passe '%c%c'" % (chr(i), chr(j))
                open('binaire', 'w').write(d)
                return
```

Une fois lancé, on obtient le résultat

```
>>> bruteforce()
[+] Binaire trouvé!
[+] Début du mot de passe 'O7'
```

Les 2 premiers caractères du mot de passe retrouvés en déchiffrant le message GPG (O7huQcYzHEPSq82m) coïncide avec ceux retournés ici.

Le binaire est disponible dans le répertoire courant sous le nom de “binaire” :

```
$ sha1sum binaire
c464f747c721e8f345016ab8696efa02b8e317ad  binaire
$ file binaire
binaire: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically
linked (uses shared libs), corrupted section header size
```

Il reste à pousser le binaire dans l'émulateur (ou à rentrer les coordonnées GPS dans Secret et fournir un mot de passe de 16 à 23 caractères commençant par O7).

```
# cd /data/data/com.anssi.secret/files
# chmod 755 binaire
# ./binaire
Bravo, le challenge est terminé! Le mail de validation est : \
4284d974a8af53aa7a85fc4e956b2d84@sstic.org
```

11 Conclusion

Au final, les 2 angles d'attaques considérés dans ce document pour la résolution du challenge – l'un orienté crypto et l'autre plus orienté reverse – ont tous les deux permis de passer du temps sur des aspects techniques extrêmement intéressants.

Les solutions proposées ne sont certainement pas les seules possibles ou les plus efficaces mais elles ont permis de discuter de nombreux points techniques dans des domaines assez variés.