Challenge SSTIC 2010

Gabriel Campana gabriel@security-labs.org
Jean-Baptiste Bédrune jb@security-labs.org

3 mai 2010

Résumé

Le défi consiste à analyser la copie intégrale de la mémoire physique d'un téléphone mobile utilisant le système d'exploitation Android, l'objectif étant d'y retrouver une adresse e-mail `@sstic.org`.

Table des matières

1 Analyse de la mémoire physique

1.1 Découverte de la mémoire

La commande `file` indique que le fichier téléchargé est compressé au format `7zip` :

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

Après son extraction, une recherche des chaînes de caractères `SSTIC` et `ANSSI` dans la mémoire physique donne quelques indices :

```
% strings challv2 | grep -i 'sstic\|anssi'
--- snip ---
com.anssi.secret.apk
com.anssi.textviewer.apk
--- snip ---
```

Quelques chaînes de caractères apportent des informations supplémentaires sur la version d'Android utilisée :

```
<4>CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00093177
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
Start proc com.anssi.textviewer for activity
  com.anssi.textviewer.textviewer: pid=227 uid=10024 gids={1015}
Start proc com.anssi.secret for activity com.anssi.secret/.SecretJNI:
  pid=233 uid=10025 gids={}
Trying to load lib /data/data/com.anssi.secret/lib/libhello-jni.so
  0x43d017f8
Added shared lib /data/data/com.anssi.secret/lib/libhello-jni.so
  0x43d017f8
Bravo ! Va lancer le binaire pour voir si ça a marché
```

Deux applications Android développées par l'ANSSI (`textviewer` et `secret`) ont été lancées sur le téléphone et semblent vraisemblablement avoir un rapport avec le challenge. L'étape suivante sera donc la récupération et l'analyse de ces deux applications.

1.2 Ou comment éviter de reconstruire la mémoire virtuelle

Les fichiers `.apk` [1] sont les paquets contenant les applications Android. Chaque application Android est encapsulée dans un unique fichier au format ZIP, qui inclut le code de l'application (fichiers `.dex`), des ressources, etc.

Deux pistes sont possibles pour récupérer les applications. La première est de reconstruire entièrement la mémoire virtuelle à partir de la mémoire physique. Cette étape s'avérerait rapidement nécessaire si un nombre important d'informations devait être retrouvé dans la suite du challenge. Bien qu'élégante, la reconstruction de la mémoire virtuelle est cependant longue et fastidieuse, d'autant plus qu'aucun outil public n'est disponible pour l'architecture ARM à notre connaissance.

Une seconde piste est d'émettre l'hypothèse que le challenge réside uniquement dans l'analyse des deux fichiers `.apk` et qu'aucune autre information ne devra être retrouvée ultérieurement dans la mémoire physique. Cette méthode présente l'avantage d'être beaucoup plus rapide à effectuer. En effet, la structure d'un fichier ZIP [2] se prête particulièrement bien au forensics. Nous avons pris le pari d'utiliser cette méthode.

Un fichier ZIP possède la structure illustrée par la figure 1 page 6 :
- une section *central directory* (signature `0x04034b50`) est présente à la fin du fichier zip, indiquant entre autre la liste des dossiers et fichiers de l'archive, leurs tailles compressées et décompressées, un CRC32, etc.
- chaque fichier compressé est précédé d'un en-tête *local header* (signature `0x06054b50`) répétant ces mêmes informations.

La récupération des fichiers inclus dans l'archive `.apk` se fait en 2 étapes : la première consiste à trouver la section *central directory* pour déterminer la liste des fichiers présents dans l'archive, et la seconde à rechercher chacun de ces fichiers en mémoire. En pratique, cette méthode donne d'excellents résultats et permet de reconstruire entièrement ces deux archives, hormis le fichier `lib/armeabi/libhello-jni.so` de l'application `com.anssi.secret.apk`. La taille des pages sur Android est 4096 octets : tous les fichiers compressés à l'exception de celui-ci ont une taille nettement inférieure à la taille des pages et sont donc retrouvés directement.

La taille de la bibliothèque `lib/armeabi/libhello-jni.so` compressée est de 9472 octets, elle est donc présente sur au moins trois pages – hélas non consécutives – de la mémoire physique. La première se retrouve facilement grâce à l'en-tête du fichier compressé, et la dernière car la bibliothèque est suivie par l'en-tête du fichier compressé suivant. Il reste à trouver les deux pages restantes, par bruteforce : le CRC32 permet

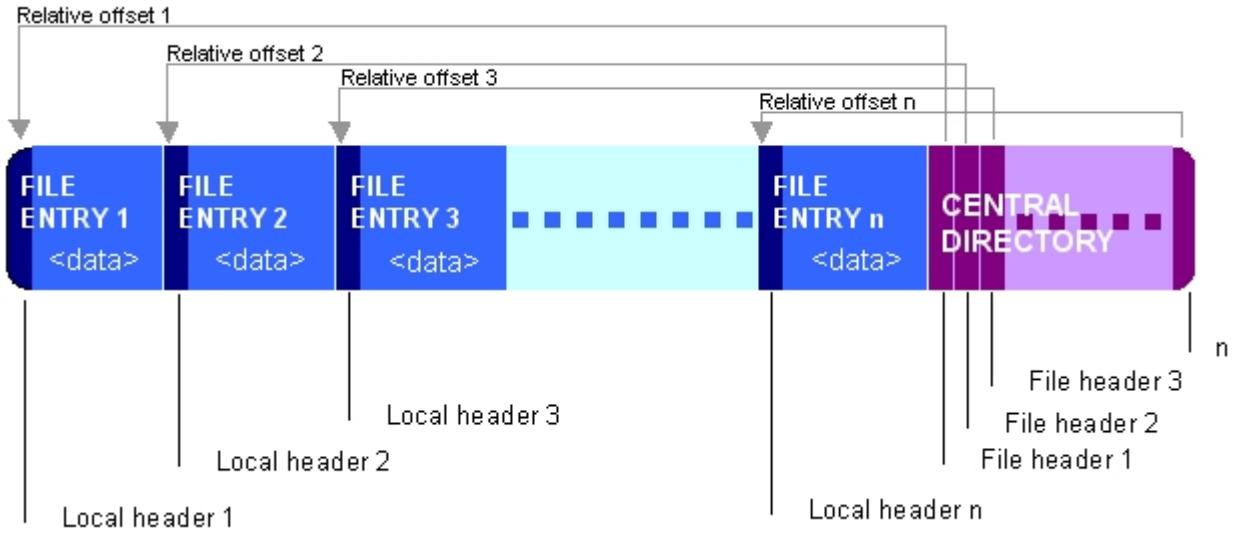

FIG. 1 – Structure d'un fichier .zip (source : Wikipédia)

de vérifier que les données décompressées sont bien celles attendues. Il est possible d'accélérer significativement le bruteforce en ne tenant pas compte des pages dont le nombre d'octets nul est supérieur à un certain ratio, afin d'éliminer directement les pages ne correspondant vraisemblablement pas à des données compressées.

Cette méthode permet de retrouver les fichiers composant les deux applications `.apk`. Le challenge va donc se poursuivre par leur analyse.

2 Émulation des applications

Google fournit un SDK [3] complet afin d'émuler le système Android. Après sa mise à jour (`android update sdk`), la création d'un virtual device avec l'API 7 sur la plateforme 2.1 se fait directement par l'interface graphique. Les deux applications s'installent ensuite avec les commandes suivantes :

```
% adb install textviewer.apk
% adb install secret.apk
```

Les figures 2 page 7 et 3 page 8 montrent les captures d'écran des deux applications émulées. `TextViewer` affiche un texte chiffré, et `Secret` présente une interface à faible affordance composée de quatre boutons *lieu* et d'un champ *mot de passe*. `Secret` crashe après avoir cliqué 4 fois sur *J'y suis !* et rentré un mot de passe invalide.

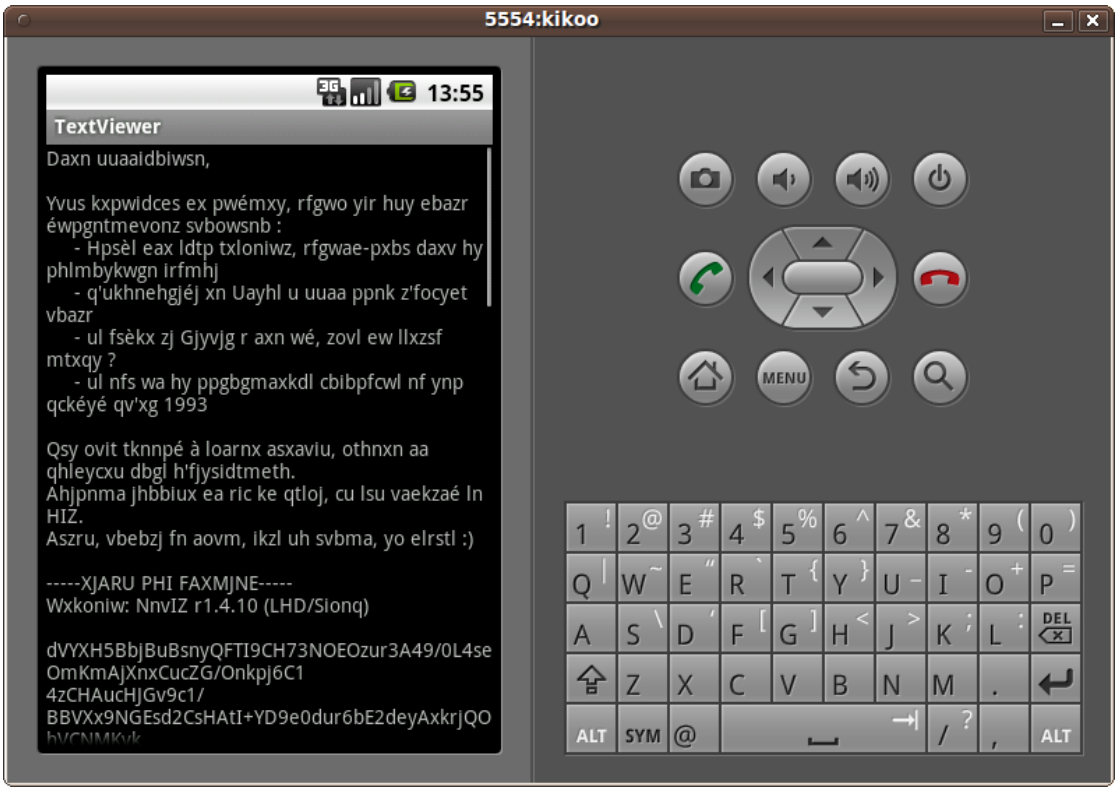

FIG. 2 – Capture d'écran de l'application `TextViewer`

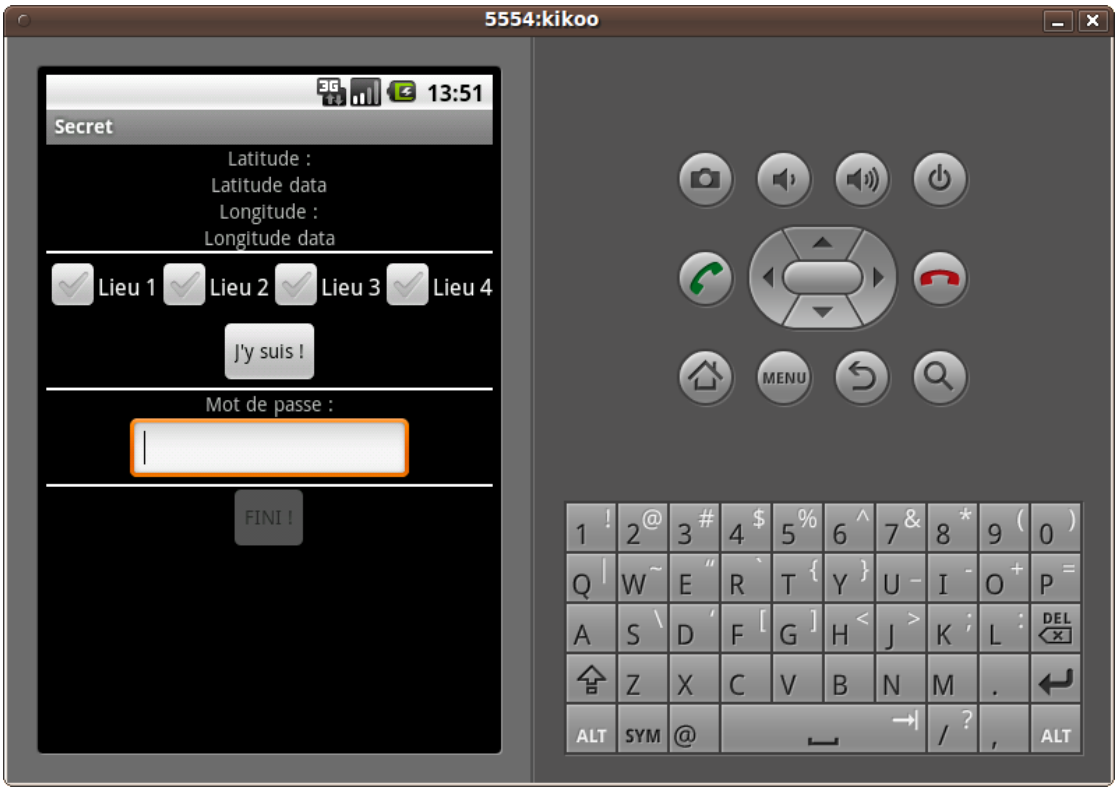

FIG. 3 – Capture d'écran de l'application `Secret`

3 TextViewer

3.1 Déchiffrement du texte

Le texte chiffré (extrait des ressources du paquet `textviewer.apk`) est le suivant :

```
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
--- snip ---
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
--- snip ---
GBKhbbN=
=8CVr
-----EOW ICU JDILJV DAD VUVCL-----
```

Un coup d'œil rapide montre que seules les lettres non accentuées semblent avoir été permutées, et la structure de deux fichiers GPG est aisément reconnue. Ces éléments laissent penser qu'un chiffrement simple a été employé. Le chiffre de Vigénère [4] est rapidement découvert et cassé grâce aux textes clairs préalablement identifiés :

– `Cher`,

– -----BEGIN PGP MESSAGE----,
– -----END PGP MESSAGE-----,
– etc.

Le texte déchiffré avec la clef BTTWFUJHA est le suivant suivi d'un message chiffré avec GPG et d'une clef publique :

```
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 :)
```

3.2 Des lieux énigmatiques

La première partie du message indique comment utiliser l'application Secret : les solutions des quatre énigmes sont des lieux dont les coordonnées GPS sont à donner à l'application, tandis que le message chiffré avec GPG contient le mot de passe.

Étant donné que l'éruption du volcan islandais *Eyjafjöll* nous empêche de prendre l'avion pour nous rendre sur ces différents lieux, nous utiliserons la commande suivante pour émuler une position GPS :

```
% adb emu geo fix -1.668881 51.123032
```

3.3 Cryptanalyse du message PGP

Les champs du message et de la clé ont été analysés avec pgpdump.net. Il apparaît que :

– le message est chiffré pour deux personnes ;
– les algorithmes de chiffrement utilisés sont RSA et ElGamal ;
– la clé publique de l'expéditeur utilise ElGamal pour le chiffrement, et DSA pour la signature.

On suppose donc que la clé à attaquer est la clé ElGamal.

Recherche de la clef privée

La clé de chiffrement utilisée est une clé ElGamal de 1024 bits. La sécurité de cet algorithme repose sur la difficulté du problème du logarithme discret dans un corps fini.

Les paramètres publics de la clé sont (g, y, p) avec p premier et g générateur du groupe multiplicatif $\mathbb{Z}_p^*$. La clé privée est le logarithme discret de y relativement à g, soit l'unique entier x, $0 \leq x \leq n-1$, vérifiant : $y = g^x \mod p$.

On appelle ordre de $\mathbb{Z}_p^*$ le nombre d'éléments de $\mathbb{Z}_p^*$, soit $p-1$. Cet ordre doit contenir au moins un grand facteur premier pour assurer une sécurité correcte à ElGamal. Afin de vérifier cette propriété, on factorise $p-1$. Or, le nombre utilisé s'avère très friable : il est composé de 21 facteurs premiers, tous de petite taille. La factorisation a été réalisée l'algorithme de factorisation par courbes elliptiques de Lenstra [5], particulièrement efficace sur des nombres composés de petits facteurs. Une applet Java publique [6] a été utilisée.

Le plus grand des facteurs premiers obtenus a une taille de 62 bits.

Calcul du logarithme discret Un algorithme de calcul de logarithmes discrets est particulièrement adapté dans les groupes multiplicatifs d'ordre friable : il s'agit de l'algorithme de Pohlig-Hellman [7], reposant sur le théorème des restes chinois. Il consiste à calculer, connaissant la factorisation de l'ordre $p-1 = p_1 p_2 \dots p_{21}$, les valeurs x_i telles que $x_i = x \mod p_i, 1 \leq i \leq 21$. Le théorème des restes chinois donne directement le logarithme discret final x.

Pour cela, on calcule les logarithmes discrets $x_i, 1 \leq i \leq 21$ tels que $y_i = g_i^{x_i} \mod p$, avec $y_i = y^{(p-1)/p_i}$ et $g_i = y^{(p-1)/p_i}$. Tous les y_i sont d'ordre p_i, conséquence directe du petit théorème de Fermat.

Chacun de ces logarithmes discrets a été calculé avec l'algorithme rho de Pollard [8]. Il s'agit d'un algorithme de Monte Carlo, reposant sur le paradoxe des anniversaires. Son principe général est de trouver une collision $X_i = X_j$ dans la séquence des $X_i = g^{a_i} y^{b_i}$, avec $X_{i+1} = f(X_i)$ où f est une fonction paraissant aléatoire.

Cette collision donne directement la valeur du logarithme discret : $g^{a_i} y^{b_i} = g^{a_j} y^{b_j}$ donc $y = g^{\frac{a_j - a_i}{b_i - b_j}}$. D'après le paradoxe des anniversaires, la complexité temporelle pour obtenir un cycle est de $\sqrt{\frac{\pi N}{2}}$ itérations, soit $\mathcal{O}(\sqrt{N})$ avec N l'ordre du groupe. Dans le

cas étudié, on pourra donc espérer calculer le plus gros logarithme discret en $\sqrt{\frac{\pi 2^{62}}{2}} \approx 2^{31.3}$ itérations, ce qui est largement faisable sur un ordinateur classique.

Mise en œuvre Le calcul a été effectué avec un programme personnel. Une applet Java est disponible sur le site utilisé pour la factorisation, mais elle est assez lente. Le programme développé ne comporte pas d'optimisations particulières. L'algorithme de recherche de cycles utilisé est l'algorithme de Brent. La marche aléatoire est celle initialement proposée par Pollard. Les multiplications dans la fonction d'itération sont réalisées avec l'algorithme de Montgomery. Les calculs sont effectués par OpenSSL, le programme est compilé en 64 bits.

Le programme a réalisé 5788258450 itérations en 1h45 avec un processeur i7 950 : aucune optimisation n'était donc nécessaire.

La clé privée est :

```
x = 01A3 477C452A DB425917 FE79D072 CC4F1EFC 27175BD9 A253C41F
F6D5C80C 7275A907 1B6DE6D6 EEDF2DF1 330CE6B3 A09805A0 492CF0D7
05F42F99 AE91680C D60F42AD 1AA929C4 8D805A3A 44E48E1A 4E908D66
F328AB33
```

Déchiffrement du message

L'approche choisie a été de créer une fausse clé privée contenant la clé secrète, et de déchiffrer le message avec GnuPG.

La fonction de génération de clés ElGamal de GnuPG a été modifiée, afin de forcer les paramètres (g, x, p) utilisés. Il s'agit de la fonction `generate` du fichier `cipher/elgamal.c`.

GnuPG utilise l'identifiant de clé présent dans le message chiffré pour vérifier qu'une clé secrète correspondant au message est bien présente dans le magasin de clés. Il est possible d'outrepasser cette vérification, en testant toutes les clés secrètes présentes dans le magasin, en passant l'argument `--try-all-secrets` à GnuPG.

On obtient alors le mot de passe :

```
$ ./gpg --decrypt --try-all-secrets message.asc
O7huQcYzHEPSq82m
```

4 Secret

Puisque les solutions des énigmes ne sont pas évidentes à trouver, il semble judicieux de comprendre le fonctionnement de l'application `Secret` supposée vérifier les coordonnées GPS de ces lieux et le mot de passe.

4.1 Reverse de classes.dex

Les paquets Android `.apk` contiennent un programme au format `.dex` (*Dalvik EXecutable*). Les fichiers `.dex` sont des programmes Java compilés en `.class` puis convertis dans un bytecode interprété par la machine virtuelle Dalvik [9].

Quelques outils sont disponibles pour désassembler les fichiers `.dex`, par exemple `Dedexer` [10] ou `smali` [11]. Le code assembleur généré permet de comprendre grossièrement le fonctionnement de `Secret`, composée des deux classes `com.anssi.secret.RC4` et `com.anssi.secret.SecretJNI`.

com.anssi.secret.RC4

La classe `RC4` est composée de cinq méthodes :
- `void RC4(key)`,
- `static void com(key, data)`,
- `array crypt(data)`,
- `void cryptself(data)`,
- `byte getByte()`.

Une analyse superficielle de l'assembleur laisse à penser que cette classe effectue bien un chiffrement RC4. Cependant, le fait que la méthode `com` soit déclarée *static* et le nom particulier de la méthode `cryptself` attirent l'attention.

com.anssi.secret.SecretJNI

La classe `SecretJNI` est composée de 11 méthodes, nous ne détaillerons que le pseudo-code des méthodes les plus.

La méthode `clinit` charge la bibliothèque native `hello-jni`. Notons la dédicace à newsoft dans la variable `coincoin` ;) :

```
static void clinit() {
    String coincoin = "bmV3c29mdCwgdHUgZXMgaW50ZXJkaXQgZG"
                      "UgY2hhbGxlbmdlIHBvdXIgc29j\naWFsI"
                      "VuZ2luZWVyaW5nIGV4Y2Vzc2lmLg==";
```

```
    String programme = "477689b3cb25eba2b9d671cb4a256c07...";
    java.lang.System.loadLibrary("hello-jni");
}
```

dechiffer déchiffre la variable programme en fonction de la clef passée en paramètre :

```
private byte[] dechiffrer(String clef) {
    byte[] v0 = hexStringToByteArray(programme);
    RC4 v2 = RC4(clef.getBytes())
    return v2.crypt(v0);
}
```

onClick est le gestionnaire d'événement associé aux clics sur les boutons. L'action associée au clic sur *FINI !* appelle la fonction deriverclef sur le mot de passe et les coordonnées GPS de chaque lieux afin d'obtenir une clef. Cette clef est passée en paramètre à la méthode dechiffrer, et le résultat est écrit dans un fichier appelé *binaire*. Finalement, une pop-up contenant le message *Bravo ! Va lancer le binaire pour voir si ça a marché* est affichée.

```
public void onClick(android.view.View v) {
    double[] lieux;
    ...
    if (condition) {
        String clef = deriverclef(mdp_edit.getText().toString(),
            lieux);
        byte[] programme_dechiffre = dechiffrer(clef);
        java.io.FileOutputStream fp = openFileOutput("binaire", 0)
            ;
        fp.write(programme_dechiffre)
        fp.close()

        String msg = "Bravo ! Va lancer le binaire pour voir si ca
             a marche";
        android.content.Context ctx = getApplicationContext(msg,
            1);
        ctx.show();
        return
    }
    ...
}
```

L'analyse du fichier classes.dex nous a appris qu'une variable programme contient le code d'un programme chiffré en RC4. La clef pour déchiffrer ce binaire est dérivée par la fonction deriverclef de la bibliothèque libhello-jni.so en fonction des coordonnées GPS et du mot de passe donnés sur l'interface graphique. Le mot

de passe est connu, il reste à analyser la bibliothèque pour déterminer les coordonnées GPS des lieux.

4.2 Reverse de la bibliothèque

La bibliothèque est analysée avec l'incontournable IDA Pro. Seul le nom de la fonction `Java_com_anssi_SecretJNI_deriverclef` est en clair, le nom des autres fonctions est obfusqué.

La chaîne de compilation croisée fournie dans le NDK [12] se révèlera utile pour déboguer (`arm-eabi-gdb`) la bibliothèque et éventuellement compiler (`arm-eabi-gcc`) des programmes ARM, par ailleurs `gdbserver` est installé par défaut sur l'émulateur :

```
% adb forward tcp:1234 tcp:1234
% adb shell gdbserver :1234 --attach 210
```

Résolution des pointeurs de fonctions

Afin d'analyser les résultats de la bibliothèque en fonction des entrées, nous avons eu l'idée de créer un programme lié à celle-ci pour appeler la fonction `deriverclef` indépendamment de l'application `Secret`. La documentation JNI [13] indique que les fonctions appelées par la machine virtuelle Java doivent respecter la signature suivante :

```
void JNICALL Java_ClassName_MethodName(JNIEnv *env, jobject obj);
```

Il apparaît rapidement qu'il est difficilement possible de créer ce programme, puisque de nombreux appels de fonctions sont effectués sur des pointeurs de la structure `env`, comme illustré par la figure 4.

Les offsets des fonctions de la structure `JNINativeInterface` sont définis dans le fichier `jni.h` et donnent une première idée de ce que fait le programme. La figure 5 page 16 montre la liste des fonctions appelées dans leur ordre d'appel. La fonction `deriverclef` est clairement divisée en deux parties : la première effectue des calculs sur les lieux et le mot de passe donné en entrée, tandis que la seconde appelle une méthode Java statique (`CallStaticVoidMethod`) pour ensuite générer la chaîne de caractères retournée.

```
LDR     R2, [R7]
MOVS    R3, 0x2F8
LDR     R3, [R2,R3]
MOVS    R0, R7          ; env
MOVS    R1, R4          ; lieux
MOVS    R2, #0          ; isacopy
BLX     R3              ; GetDoubleArrayElements
```

FIG. 4 – Appel de la fonction env->GetDoubleArrayElements

Offset	Fonction
0x2a4	GetStringUTFChars
0x2c0	NewByteArray
0x2f8	GetDoubleArrayElements
0x340	SetByteArrayRegion
0x24	FindClass
0x1c4	GetStaticMethodID
0x234	CallStaticVoidMethod
0x320	GetByteArrayRegion
0x44	ExceptionOccurred
0x2a8	ReleaseStringUTFChars
0x29c	NewStringUTF

FIG. 5 – Offset des fonctions appelées

Analyse du code

La première partie de la fonction récupère les arguments de la fonction (les coordonnées GPS et le mot de passe) :

```
void Java_com_anssi_SecretJNI_deriverclef(
          JNIEnv *env, jobject jobj, jdouble *a1, jstring a2) {
    char sig[8], data[256], hash_coord[8], buf[32], result[65];
    jbyte *array_0x20;
    jdouble *coord_gps;
    char *p, *password;
    int i;

    // recuperation du mot de passe
    password = env->GetStringUTFChars(env, a2, 0);
```

```
// initialisation de data
G4Cy(data, 256);

array_0x20 = env->NewByteArray(env, 32);

strcpy(sig, "([B[B)V");

// recupere les coordonnes GPS
coord_gps = env->GetDoubleArrayElements(env, a1, 0);
```

Un hash `hash_coord` est ensuite créé à partir des coordonnées :

```
for (i = 0; i < 16; i += 2) {
    x, y = SXXJZ(coord_gps[i], coord_gps[i+1]);
    hash_coord[i] = i3IkHSwJkop7(x, y);
}
```

Puis une chaîne de caractères `buf` est créée à partir du mot de passe et de `hash_coord`, par les fonctions `8j3zIX` et `sdlHj` :

```
// mise a jour data en fonction du password
8j3zIX(data, password, strlen(password), 0);

// mise a jour de data en fonction du hash des coordonnees
8j3zIX(var_1B4, hash_coord, 32, 0);

// creation d'un buffer en fonction data
sdlHj(data, buf);

SetByteArrayRegion(array_0x20, 0, 32, buf);
```

Le pseudo-code responsable de l'appel de la méthode statique est le suivant :

```
char *classname;
char *s = "\xF4\x94\x7D\x75" \
          "\x17\xEE\x04\xFB" \
          "\xFE\xD4\x63\x3F" \
          "\x15\xF2\x12\xFC" \
          "\xB8\x00\x00\x00";

// construction du nom de la classe
for (i = 0; i != 0x11; i++) {
    classname[i] = hash_coord[i & 0x7] ^ s[i];
}

// construction de la fin du nom de la classe
// et du nom de la methode
```

```
    classname[0x11] = classname[0] ^ 0x31;
    classname[0x12] = classname[1] ^ 0x2c;
    classname[0x13] = classname[2] ^ 0x59;
    classname[0x13] = classname[3] ^ 0x2f;

    class = env->FindClasse(env, classname);
    if (class != NULL) {
        classname[3] = '\x00';
        method_id = env->GetStaticMethodId(env, class, classname,
           "([B[B)V");

        if (method_id != NULL) {
            env->CallStaticMethod(env, class, method_id,
               array_0x20, array_0x20);
            env->GetByteArrayRegion(env, array_0x20, 0, 32, buf);
        }
    }
```

La troisième argument de la fonction `GetStaticMethodId` est le nom de la classe à appeler. Nous avons vu précédemment que la classe `RC4` possède une méthode statique `com` et ailleurs, la signature `"([B[B)V"` passée en argument correspond bien (`static void com(byte[], byte[])`). En partant de cette hypothèse, la construction du nom de la méthode montre que `"com\x00"` $\oplus$ `"\x31\x2c\x59\x2f"` = `"RC4/"`. Nous pouvons donc en déduire que le nom de la classe est `com/anssi/secret/RC4`, et le nom de la méthode appelée est bien `com`.

Étant donné que `s` est connue (copie du tableau d'entiers `dword_36E0` au début de la fonction), nous alors déduire la valeur de `hash_coord` avec les quelques lignes de python suivantes :

```
hash_coord = ''
classname  = 'com/anssi/secret/'
s          = '\xF4\x94\x7D\x75\x17\xEE\x04\xFB'

for i in range(0, 8):
    hash_coord += chr(ord(classname[i]) ^ ord(s[i]))
```

Nous connaissons donc la valeur de la variable `hash_coord` dérivée des coordonnées GPS. Cette valeur, ainsi que le mot de passe, est ensuite utilisée par les fonctions `8j3zIX` et `sdlHj` pour créer un buffer de 32 octets passée en argument à `com.anssi.secret.RC4.com`.

Finalement, la clef finale est retournée en hexadécimal :

```
    env->ExceptionClear(env);
```

```
    // construction de la chaine en hexadecimal
    p = result;
    for (i = 0; i < 32; i++) {
        sprintf(p, "%02x", buf[i]);
        p += 2;
    } while (r4 != r6);
    result[64] = '\x00';

    env->ReleaseStringUTFChars(env, a2, password);

    return env->NewStringUTF(env, result);
}
```

Nous avons donc tous les éléments en main pour déchiffrer le programme, sans connaitre les coordonnées GPS originales, et sans avoir analysé d'autres fonctions que `deriverclef`.

5 Dernière étape

Connaissant cette valeur et le mot de passe, il ne reste plus qu'à exécuter `Secret` en modifiant la valeur après le calcul fait sur les coordonnées avant son utilisation, avec `arm-eabi-gdb` par exemple :

```
% gdb -q
gdb> target remote tcp 127.0.0.1:1234
gdb> b *0x80a017ec
gdb> Breakpoint 1 at 0x80a017ec
gdb> c
gdb> ...
gdb> Breakpoint 1, 0x80a017ec in ??()
gdb> set *(int *)$r8 = 0x5a10fb97
gdb> set *(int *)($r8+4)= 0x88778076
gdb> c
```

Le binaire est alors créé, il ne reste plus qu'à l'exécuter pour obtenir le mail de validation :

```
% adb shell /data/data/com.anssi.secret/files/binaire
Bravo, le challenge est terminé !
Le mail de validation est : 4284d974a8af53aa7a85fc4e956b2d84@sstic.org
```

Remarquons au passage une dernière difficulté rencontrée dans le programme généré : une vérification est effectuée sur le nom du programme (`argv[0]`) exécuté :

```
int main(int argc, char *argv[]) {
    char *mail = "42849d74a8af53aa7a85fc4e956b2d84@sstic.org";
    char *msg  = "Bravo, le challenge est termine ! Le mail "\
                 "de validation est : %s\n";
    char a, b, c;
    int n;

    n = strlen(argv[0]);
    c = argv[0][n-1];

    // permutation de 2 caracteres
    a = mail[c - 0x61];
    b = mail[c - 0x60];
    mail[c - 0x60] = a;
    mail[c - 0x61] = b;
```

```
    printf(msg, mail);

    return 0;
}
```

Si la dernière lettre du programme exécuté n'est pas le caractère e, l'adresse e-mail générée est invalide. Ainsi, l'adresse e-mail n'apparaît pas en clair dans les strings du programme. Vérification simple, mais légèrement perturbante lorsque l'on croit enfin avoir résolu le challenge mais que le binaire a été renommé.

FIG. 6 – Déchiffrement du programme et création du binaire : challenge résolu !

6 Conclusion

La solution présentée est fonctionnelle mais des questions restent en suspens : les solutions de ces énigmes saugrenues et les coordonnées GPS associées ne sont pas connues. L'hypothèse de départ comme quoi la mémoire physique sert uniquement à retrouver les deux applications `.apk` est heureusement vérifiée. Il aurait sinon été nécessaire de reconstruire la mémoire virtuelle, qui aurait été bien plus difficile et aurait nécessité plus de temps.

Je tiens à remercier grandement jb pour la résolution de la partie cryptographique. Et enfin, merci à l'ANSSI d'avoir réalisé ce challenge original et amusant à résoudre :-)

"I may be Paranoid, But not an Android !"

Références

[1] APK (file format)
http://en.wikipedia.org/wiki/APK_%28file_format%29

[2] ZIP (file format)
http://en.wikipedia.org/wiki/ZIP_%28file_format%29

[3] Android SDK
http://developer.android.com/sdk/index.html

[4] Chiffre de Vigenère
http://fr.wikipedia.org/wiki/Chiffre_de_Vigen%C3%A8re

[5] H. W. Lenstra, Jr., *Factoring integers with Elliptic Curves.* Ann. Math. 124, 649-673, 1987.

[6] Factorization using the Elliptic Curve Method
http://www.alpertron.com.ar/ECM.HTM

[7] S. Pohlig and M. Hellman, *An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance*, IEEE Transactions on Information Theory (24) : 106–110, 1978

[8] J. Pollard, *Monte Carlo methods for index computation mod p*, Mathematics of Computation, Volume 32, 1978.

[9] Dalvik VM Internals
http://sites.google.com/site/io/dalvik-vm-internals

[10] Dedexer – Disassembler tool for DEX files
http://dedexer.sourceforge.net/

[11] smali – An assembler/disassembler for Android's dex format
http://code.google.com/p/smali/

[12] Android NDK
http://developer.android.com/ndk/index.html

[13] Java Native Interface
http://en.wikipedia.org/wiki/Java_Native_Interface