A solution to SSTIC 2010 Challenge

Arnaud Ebalard <arno@natisbad.org>

June 4, 2010

Abstract

The goal of SSTIC 2010 Challenge was to analyse the complete physical memory dump of an Android phone, in order recover a hidden email address ending with `@sstic.org`.

After an initial step involving the rebuild of two Android application packages (APK) available in a fragmented form in memory, a first "funny" crypto part provides access to instructions. Then, in the second part of the challenge, the player is given a choice between two different paths to solve the challenge.

The first path involves finding the answers to four riddles (two of those are complex) in order to get access to a set of four GPS coordinates and then solve a "serious" cryptography problem to recover a password encrypted in a GPG message.

The second path involves reverse engineering a library (ARM assembly) and an Android DEX file, in order to recover the previously discussed GPS coordinates and password.

The technical details associated with the two paths are presented in the document.

Contents

1 Introduction

As discussed on SSTIC 2010 Challenge page (in french), the purpose is to analyze the complete physical memory dump of an Android phone in order to recover a hidden email address ending with `@sstic.org`.

This document provides details on the leads followed and the results obtained in order to recover that address. It is a translation by its author of the original solution written in french. The date of publication of the initial solution has been kept for that translation.

Except otherwise specified in the document, all the steps have been performed on a Debian GNU/Linux (x86 32-bit) system.

Various files are embedded in this document. Based on the PDF reader of choice, here is how you can access those:

- If you are reading this document using Adobe Reader, you can access those files via "View" > "Navigation Panels" > "Attachments".
- If you are using Evince, you can select "Attachments" at the top of the side pane instead of the usual "Index". If not already displayed, the side pane is available via the "View" menu or using F9 shortcut.
- Otherwise, you can use `pdftk this_document.pdf unpack_files` to extract all embedded files in current directory.

2 Initial memory analysis

The first step consists in downloading the memory dump:

```
$ 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
```

The archive contains a 100Mo LZMA-compressed file (challv2). Extraction is performed using 7z:

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

A quick analysis of the strings found in the file (using **strings**) confirms the the system is running 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
```

Obviously, even if the data contain lots of email addresses (and many `@` characters in general), no address ending with `@sstic.org` is directly available in the memory dump:

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

The `-e` option of **strings** can be used to extract unicode strings from the dump. Another run with the keywords associated with the SSTIC and ANSSI[1]:

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

An hexadecimal editor (**hte** for instance) can be used to confirm the fact that those strings are embedded in X.509 certificates found in the memory dump.

The lookups for keywords ('anssi' among others) in the strings available in memory indicate various references to 2 Android application packages (APK), probably developed by the ANSSI for the challenge:

[1]SSTIC 2010 Challenge was created by the French Agence Nationale de la Sécurité des Systèmes d'Information

```
$ 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
```

An exhaustive reading of the strings returned by **strings** (over 600000) is a painful but (a posteriori) rewarding task: it provides additional information which may help in the rest of the challenge.

The lead associated with the 2 applications (**com.anssi.secret** et **com.anssi.textviewer**) probably developed by the ANSSI is promising and is worth following at first.

3 Recovery of Android applications (.apk)

3.1 Short introduction to Android

The Android operating system, developed by Google, has a specific architecture, different from the ones found in common OS and distributions also based on the Linux kernel. Good introductions to Android architecture are available here and here.

Some important points regarding the system to be aware of in the context of the challenge are:

- Android is based on Linux kernel. Among other things, it handles hardware support, power and **memory** management.
- Libraries (SSL, libc, . . .) available in userland are used by applications to get access to high level services. It is interesting to note that the system's libc known as **bionic** is not the usual libc found on most Linux systems.
- Android targets (at the moment) embedded devices such as mobile phones and is available only for **ARM-powered** devices.
- Android applications do not have direct access to the processor of the device. They run on a system-specific virtual machine: Dalvik. Applications

are developed in Java and compiled to a specific bytecode understood by Dalvik (different from standard Java bytecode). Compiled applications are in **dex format**.

- Applications are provided **as packages (i.e. APK)**. APK file format is similar to JAR. **It's a ZIP archive** including a .dex, resources files, a manifest file, an X.509 certificate, ...

The content of an unzipped APK is given below:

```
$ 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 Alternatives for rebuilding APK

The two Android applications developed for the challenge have potentially been installed on the system from which the memory dump was taken.

It seems the packages for those applications have been loaded in memory (probably during install) and are still in it (at least partially). In order to continue, we perform the recovery of those applications from the memory dump.

The main challenge associated with that task directly results from the fragmentation by the Linux kernel: the physical memory is sliced into 4Ko pages; the kernel provides applications a different view called (virtual memory). A file mapped linearly in virtual memory is spread on various non contiguous pages in physical memory.

In order to recover our two Android applications packages, at least two alternatives exists:

1. Find internal structures used by Linux Kernel to manage memory and use those to reconstruct the virtual address space of system processes.

By giving access to portions of virtual memory, this approach provides a linear view of mapped files, independendtly of their type. Nonetheless, it will be effective only for files mapped in memory by processes running when the memory dump was performed.

2. Consider the memory dump as a puzzle made of 4Ko pieces and use the specificities of ZIP files structure to perform the reassembly. The main drawback of this method is that it is not generic and requires understanding the structure of the file to recover (ZIP in our case). The main advantage compared to previous method is that it may allow the recovery of files available in freed pages associated with old (no more running) processes.

Considering the goal is the recovery of a unique type (APK), the second alternative has been used by the author.

APK being simple ZIP archives, next subsection provides an introduction to this file format. Implementation details and problems encountered during reconstruction are covered in the subsection after.

3.3 ZIP file format

A good introduction to ZIP file format is provided on associated Wikipedia page. Additional details are also available here.

A ZIP archive is made of the three following main types of elements:

1. **ZIP End of Central Directory Record**: this element is located at the end of the archive and provides information on the elements directly preceding it (**ZIP Central Directory File Headers**), which make up with it the Central Directory (a kind of table of content for the archive). Among provided information, **ZIP End of Central Directory Record** gives the size of Central Directory and the number of **ZIP Central Directory File Headers** it contains. **ZIP End of Central Directory Record** starts with a 4 bytes signature value 0x06054b50.

2. **ZIP Central Directory File Header**: various entries of this type are found before the **ZIP End of Central Directory Record**. Each of those provides details on a specific file contained in the archive: CRC32 of the uncompressed version, size of the compressed version, size of the uncompressed version, filename. But this entry also provides the position of **ZIP Local File Header** in the archive (the compressed file with a leading header). A **ZIP Central Directory File Header** starts with the 4 bytes signature value 0x02014b50.

3. **ZIP Local File Header**: each file (or folder) in the archive is associated with such an entry. Among other things, it provides information on the compression method, modification date, CRC32 (of uncompressed version), size of compressed and uncompressed file, filename, . . . A **ZIP Local File Header** starts with the four byte signature value 0x04034b50.

Following picture [2] borrowed to previously cited Wikipedia article provides a graphical overview of ZIP file format

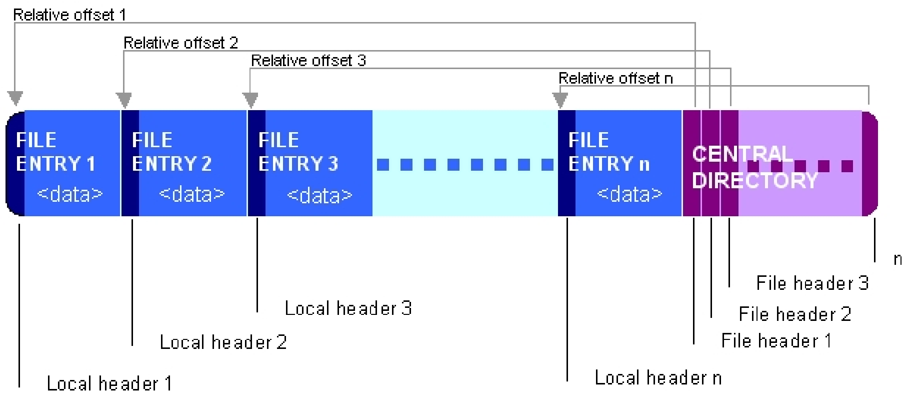


Figure 1: ZIP file format

3.4 Technical details on APK recovery

As discussed previously and using the information provided on ZIP file format, this subsection offers some technical details on APK recovery from the memory dump. A Python module (**zipfinder.py**) providing the code used for that purpose is embedded in the document.

The first step consists in scanning the memory dump looking for the signature value associated with **Zip End of Central Directory Header** (0x06054b50). This element contains the total size of Central Directory, i.e. the size of all **ZIP Central Directory Headers** preceding it.

If this piece of information positions the first Central Directory Header of this ZIP archive in the page in which the **Zip End of Central Directory Header** has been found, then all the information about the files in the archive is likely to be available in it. If this is not the case (indicating the Central Directory is spread over at least two pages), it is not worth trying to reconstruct the associated archive. This decision is motivated by the fact that the 2 APK we are interested in are not part of archives with fragmented Central Directory.

Then, each ZIP Central Directory File Header from the archives to be recovered is used to create a list of the files contained in these archives. For each of those files, their name, CRC32, compressed and uncompressed size can be used to discriminate them.

The next step consists in a second pass on the memory dump, looking this time for the 4 bytes signature value of **ZIP Local File Headers**. If the CRC32 value, filename, compressed and uncompressed sizes (all available in this ele-

[2]available at http://en.wikipedia.org/wiki/File:ZIPformat.jpg, under a Creative Commons Attribution-ShareAlike 3.0 licence

ment) do match those in the file to recover, the recovery work described below is performed. Otherwise, this element is not considered.

All the structural elements of the ZIP file considered until now had little chance to be fragmented considering their limited size. Among the files to recover some are larger than the size of a page. Fragmentation is guaranteed for those files.

Nonetheless, for the two archives which matter, those fragmented compressed files are not that long (at most 4 pages). For instance, the compressed version of **lib/armeabi/libhello-jni.so** file belonging to one of the APK is 9472 bytes long. This compressed element is spread on 4 pages.

For a given compressed file, the idea is to test all the possible combinations of the required number of pages. If decompression is then successful and the CRC32 value matches the one found in the **ZIP Local File Header**, this validates current combination of pages.

The complexity of this method is exponential in the number of pages on which the compressed file is fragmented. The total number of pages in the memory dump (100663296/4096 = 24576) makes this method unsuitable without additional optimizations. The two following facts can be used to reduce the complexity:

- *The central pages of compressed files are all made of compressed data (by construction), which is not the case for most of the pages found in memory.* This allows separating them in a simple fashion: a page which ends up having a reduced size after compression (compared to the original page) is not worth considering. This heuristic allows creating a set of 1271 interesting pages to be used as central pages.

- *A compressed file is followed in the archive either by a* **ZIP Local File Header** *(with signature 0x04034b50) or by a* **ZIP Central Directory File Header** *(with signature 0x02014b50).* Looking for the last page only in pages containing those values limits the search to a set of 2071 pages (instead of 24576 initially).

Those optimizations allow for instance[3] to get a reduced complexity of $24576^3 = 14843406974976$ to $1271^2 \cdot 2071 = 3345578311$ tests for the recovery of the compressed version of **lib/armeabi/libhello-jni.so**.

The previous value is still large (each step requires concatenations and at least a decompression test). During the development of **zipfinder.py** module which implement the heuristic described above, frequent tests have required reducing **temporarily** the set of compressed pages to the first one found. By limiting this set to the first 50 ones, the recovery of **libhello-jni.so** terminates successfully. In fact, those first 50 pages[4] are sufficient to recover the 2 APK

[3]The first page is the one containing the ZIP Local File Header

[4]it may be possible to reduce that value

(associated to TextViewer and Secret). In fact, this value allows the recovery of 7 complete APK; the procedure taking less than 4 minutes on a recent laptop. This specific value of 50 has been kept in the module embedded in this document.

```
>>> from zipfinder import *
>>> c, i = recover_zip_files("challv2")
[+] Looking for already encrypted pages
    => 50 pages in that pool                           # !! 50 first
[+] 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 ...
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
```

The 2 lists returned by recover_zip_files() respectively contain classes associated to complete and incomplete[5] archives. The only remaining step is to export APK on disk.

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

A quick look at the content of 'c' list is sufficient to infer that **com.anssi.textviewer.apk** et **com.anssi.secret.apk** are respectively the first and last ZIP files of the list. For that reason, they correspond to 00.apk and 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
```

[5] for which at least one of the embedded compressed file cannot be recovered

```
$ 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
```

The two recovered archives are both embedded in the document (along with **zipfinder.py** module).

3.5 Conclusion

The recovery of the 2 APK developed for the challenge has been performed using the specificities of ZIP file format and some simple heuristics to limit brute force duration.

This method worked mainly because of the limited size of compressed files found in the archive to recover. If those had been fragmented on a larger number of pages, this would have required to also reconstruct virtual memory or to find additional heuristics.

4 Interaction with applications

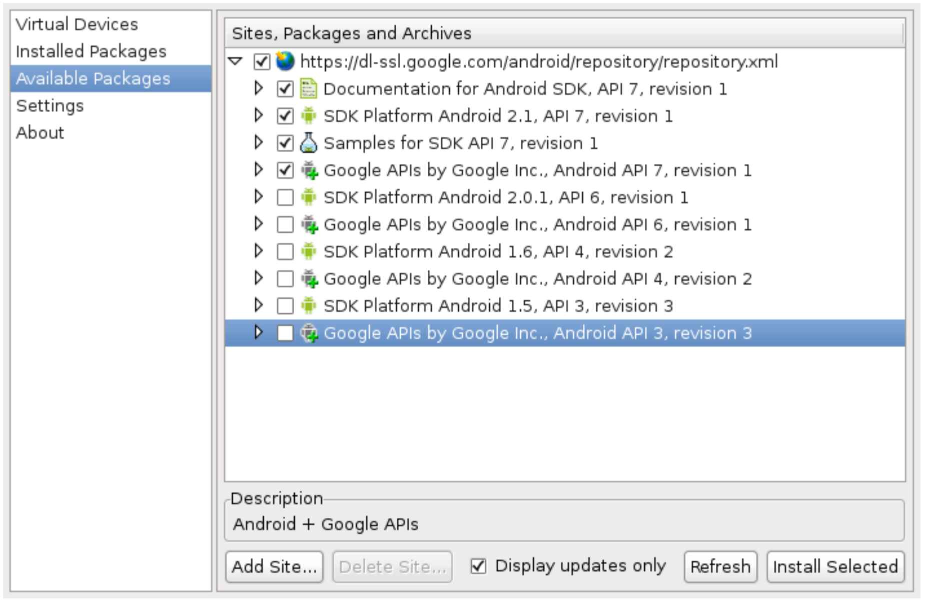
4.1 Installing Android emulator

The Android development kit (SDK) includes an emulator. It easily allows installing and testing Android applications. The SDK can be downloaded here.

Once the archive downloaded and uncompressed, the graphical application “Android SDK and AVD Manager” available in the **tools** folder can be launched.

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

In the “Available Packages” tab, you need to select the elements associated with the SDK for Android 2.1 as presented on the following picture and then perform the installation (via “Install Selected”).

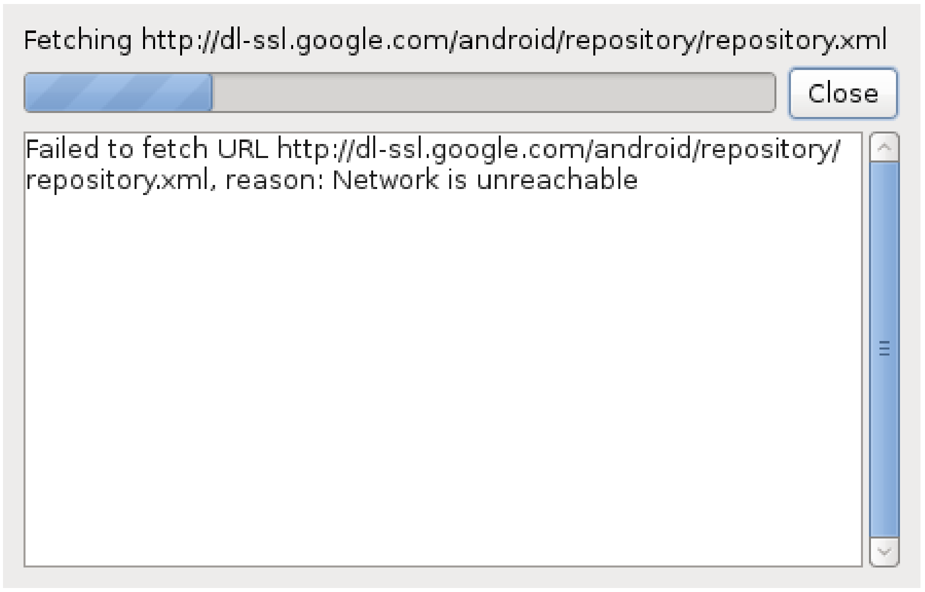


☑ https://dl-ssl.google.com/android/repository/repository.xml
- ☑ Documentation for Android SDK, API 7, revision 1
- ☑ SDK Platform Android 2.1, API 7, revision 1
- ☑ Samples for SDK API 7, revision 1
- ☑ Google APIs by Google Inc., Android API 7, revision 1
- ☐ SDK Platform Android 2.0.1, API 6, revision 1
- ☐ Google APIs by Google Inc., Android API 6, revision 1
- ☐ SDK Platform Android 1.6, API 4, revision 2
- ☐ Google APIs by Google Inc., Android API 4, revision 2
- ☐ SDK Platform Android 1.5, API 3, revision 3
- ☐ Google APIs by Google Inc., Android API 3, revision 3

Description
Android + Google APIs

Figure 2: Selecting Android 2.1 SDK elements to be installed

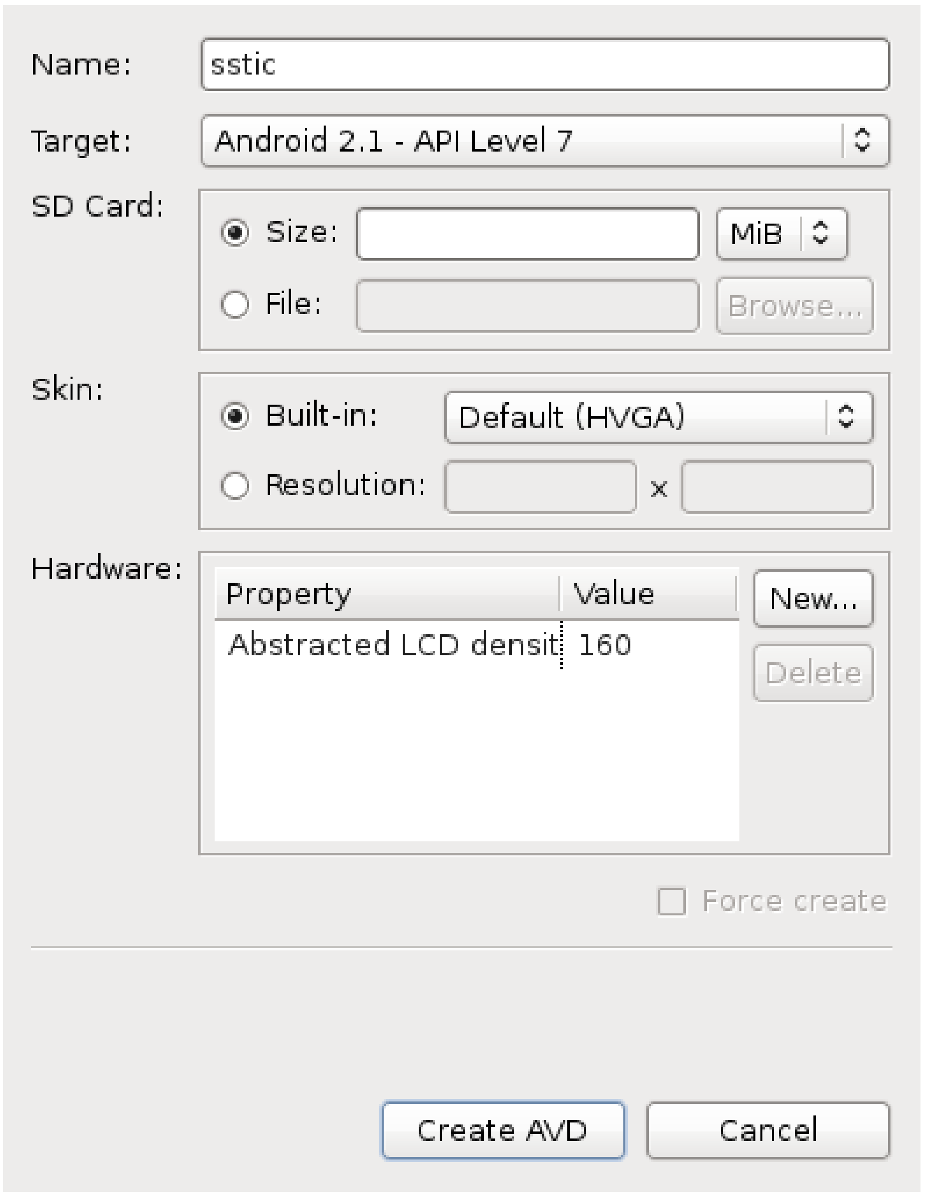
Note that if the following message appears on Debian, you simply need to set the value of `/proc/sys/net/ipv6/bindv6only` entry to `0`. The reader interested by the details of the bug can consult associated Debian bug report #560044.


Fetching http://dl-ssl.google.com/android/repository/repository.xml

```
Failed to fetch URL http://dl-ssl.google.com/android/repository/
repository.xml, reason: Network is unreachable
```

Figure 3: Debian Bug #560044

Once the elements downloaded and installed, an Android Virtual Device can be created from “Virtual Devices” tab by clicking on “New”. It needs to be given a name and target (“Android 2.1 - API Level 7”).



Name:	sstic
Target:	Android 2.1 - API Level 7
SD Card:	[✓ Size:] MiB
	File:
Skin:	[✓ Built-in:] Default (HVGA)
	Resolution: x
Hardware:	

Property	Value
Abstracted LCD densit	160

Figure 4: Creation of an Android Virtual Device (AVD)

The AVD is ready to be started and used by clicking on “Start…”. The result is presented below:

Figure 5: Running Android emulator

4.2 Installing APK in the emulator

Installing the two APK (**com.anssi.secret.apk** et **com.anssi.textviewer.apk**) using **adb** tool (found in the **tools** folder of the SDK) is done in the following way:

```
$ 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
```

The two applications are now available on the virtual device.

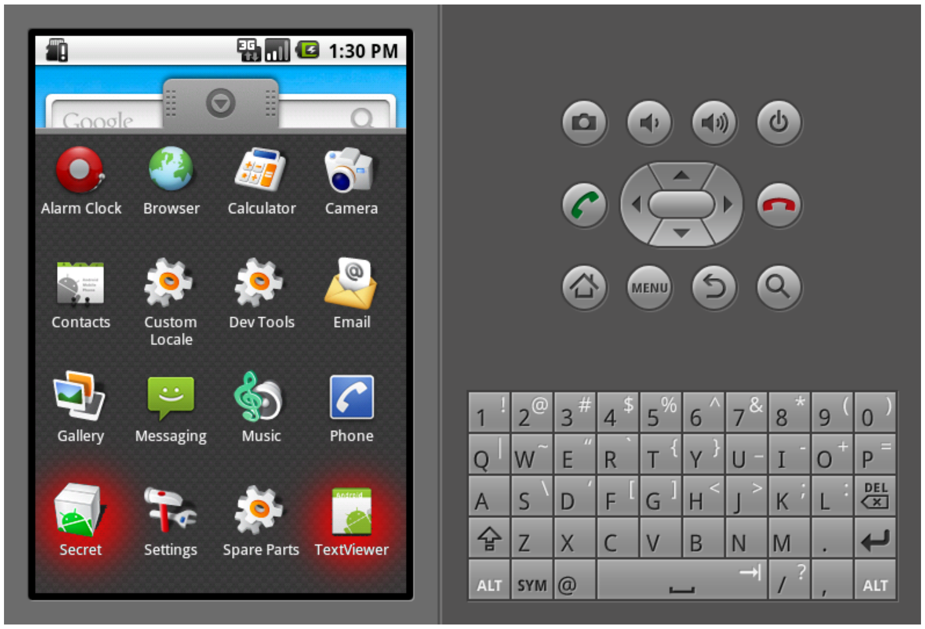


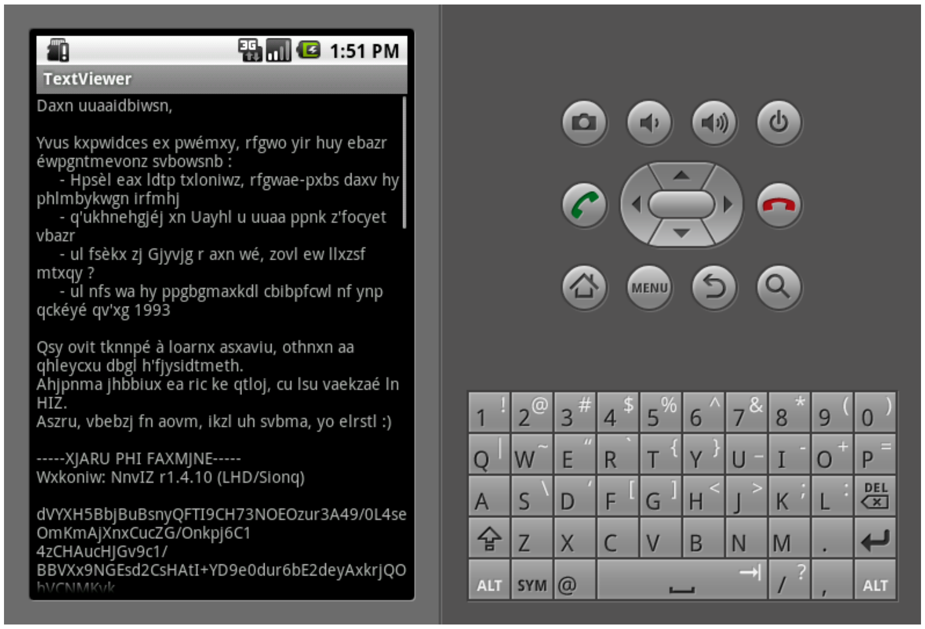
Figure 6: Secret et TextViewer applications installed in the emulator

4.3 Interacting with the applications

4.3.1 TextViewer

Once started, TextViewer application simply displays the content of the **chiffre.txt**[6] resource file found in the APK.

[6]french for encrypted.txt



```
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)

dVYXH5BbjBuBsnyQFTI9CH73NOEOzur3A49/0L4se
OmKmAjXnxCucZG/Onkpj6C1
4zCHAucHJGv9c1/
BBVXx9NGEsd2CsHAtI+YD9e0dur6bE2deyAxkrjQO
```

Figure 7: TextViewer application, once launched

The work on the **chiffre.txt** file is detailed in the three following section. It is divided in two main steps:

- a first simple step consisting in the recovery of the clear text associated with the encrypted version found in the file. Recovered text provides instruction on how to use Secret application.
- a second step of cryptography detailed later in the document.

The work can be performed offline (outside the emulator), directly on **chiffre.txt**.

4.3.2 Secret

Secret application waits for 4 GPS positions and a password.

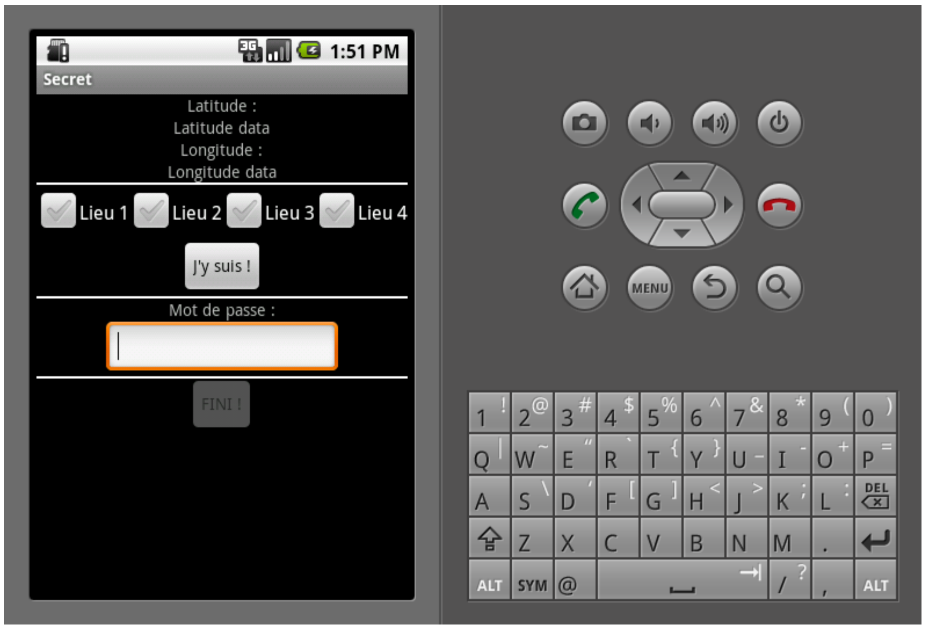


Figure 8: Secret application, once launched

The Android emulator allows passing GPS coordinates via the console available on `localhost:5554` using **telnet**. The syntax to change the latitude and longitude value is provided below. To pass a -1.64 degrees longitude value and a 48.11 degrees latitude value:

```
$ 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
```

The changes are immediately reflected in the application. This short description provides the basis of the interactions with the application.

4.4 Using GDB with Android emulator

To simplify the understanding of Secret's internal behavior in parallel with the reverse engineering of one of its component (discussed later), it is useful to be able to debug its behavior

The installation and use of **gdb** to follow the behavior of an application running in the emulator is described below. Here are the steps:

- Downloading **gdbserver** and its client for Android (from the NDK)
- Installing **gdbserver** in the emulator
- Installing a port redirection (to access it from outside the emulator)
- Attaching a process in the emulator
- Connecting to **gdb** client from oustide the emulator

Once the archive of Android NDK downloaded and uncompressed, **gdbserver** binary can be pushed on the emulator using the **adb** tool available in the **tools** folder of the 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)
```

Before starting **gdbserver**, port 1234/tcp in the emulator needs to be redirected to local port 1234/tcp on the host device. This allows debugging from outside the emulator an application running in the emulator. The port redirection can be installed using a simple command in the console of the emulator:

```
$ 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 can then be attachedin the emulator to the process to debug. **ps** can be used to recover the PID of the running application.

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

On the host, once the **gdb** client is started, the “target remote localhost:1234” command is used to connect from the host to the debugger in the emulator using the installed port redirection.

```
$ ./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 Decrypting text message (chiffre.txt)

The **chiffre.txt** file found in **com.anssi.textviewer.apk** and displayed by the application contains the following text. The file is also embedded in the document for the interested reader.

```
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)

dVYXH5BbjBuBsnyQFTI9CH73N0E0zur3A49/0L4se0mKmAjXnxCucZG/0nkpj6C1
4zCHAucHJGv9c1/BBVXx9NGEsd2CsHAtI+YD9e0dur6bE2deyAxkrjQ0hVCNMKvk
sta3nELRa4wJP4JC9BHopVT5BF8cRCX2+Aq6k9C0V8Y8QIAjKiMq9qZAg9Hg2mpW
/N7n40257FJsAunB5KkH3x0De9XHqNSs94qc08+62yCAa5uKlpzqxHVw097rRA66
E2F1ESH4748MDq0wF1NXdf0SqpUwt1R4ThYlHdNY2V0IgDnuZkbC5C0ZRMBYb38b
aJFH2wT3MnBUBqtWh5v0Htf/eEzWbBdeiLR5G3ebE/0gdNqMRpyaUaM2y70KH/0c
ZTuq+0YYGxoQaP3mM1Geic1Z+cSUHJN0pp69rCDjiibwqHiRjkRpxdcFTFv1s3nQ
xClnC04HzWo20Q1GQmXCWnPJSqGgELEab4fbHvzH2DSuFR72jmDucsxvJv8MSfWh
Cw+96H054cyHZTGk03I1Q05dbP2YUPf0V5Z8P/xh3vd82/+kh0sjvR54fDRB9t0f
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
DIdqhidda2WMszboUGkd4LYhnO6WmE2+QVzqkX/nUOR+ccX5HveRA64b1PfdZmOf
bKQtzo6pT5HGC6U3Fwx2r4dRDaC95+ejCKxXb3Aefnw2n7HIzT3iWPYBtk6oKAT7
vsMOU/45/OvDMnW4GVnAb5OxNRf5BH3VTVsOqI13gNb3cIFSXVtfXOfAB656Lv5U
A73RBF43ALXe7uegYLFrEyHjJwcRVFSjmcGGSbHCcahOMCEOOXP1ToiVVo6xTmDt
SYqEgjlhpR4iqVx1Z+JUsXnk2CqX+u7rXY5DkTZ8xahLuTyXQ7JgZAEfc29vufQl
JPsuefYofQAsQH5ouSWfIN9tZPeqEM5OkNQ+jZvAJrNJADvYWpop+8rCHpFBHKLl
NBTZYbeRIwNVUdZCJnkLVpIBUpLCIzXYK4UJJglJyd6MKfguSeWKUaLkHJNQs5bO
JKQCLd1cdwu94jCzwNOWtCvDUmv6mOyzzHvYVka9z/jRz9lqvJXJGYdn+8kRUTZz
Xe3aRrl7+cZOG2nNjmIz1URMlqTatyCaX3ww+rJTVlmUHohMA0xzI7eV4RNpj39P
UpHRNqXOF4tykU9cCWs9/qvtudsi7l2HaoXfbSCAHLTs4NkaK3u6ft3PnPgtoqJt
YQEk90G5QbHv9Nf1J0eb9B5TtZhE3OMp8MZmfywaKfHDCJJWnbW4d1JelEMP0k27
OnSMrZ84G6nT5vW36ZFWjnKZHl6Uyof8LRtrSIbeGROSpY4wLOZavMqX8zmHoboO
Q7UTtrXSLITZ8BX/cF89KBXukj/qMRWJAruiJLyM7iKx/mdBO2uikuH/IfLXEXWB
hsin7IbLoMub4Ejc95ypJKBXoWqJMpMDYZPAEpNYX6X7hXIcWTQOUS4OHABDVNG+
BxkDEH5ZQJU7JRDiotaCuHvykEEbyfZF4WElle91aaLmWXGbk0tWot0eUzVJh/cv
GBKhbbN=
=8CVr
-----EOW ICU JDILJV DAD VUVCL-----
```

The text is not directly readable but seems to have kept its initial structure: some characters have not been modified by the specific cipher used on the original file. This is for instance the case for the space but also for dashes ('-') and more generally for all punctuation marks and accented characters.

Two blocks looking like PEM-encoded X.509 certificates or ASCII armored GPG messages/keys appear in the text. This is moms likely GPG elements because the header of first message perfectly matches (from a format standpoint) the one of a common GPG message:

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

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

This remark is also valid for footers:

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

This conclusion also applies to the second message which format perfectly match one of a GPG public key:

```
Common      : -----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-----
```

It is interesting to note that the cipher used here is not a simple dictionary permutation (like rot13) because identical elements of the original text do not result in the same encrypted value. This is for instance the case of the line found below the headers of GPG elements:

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

One of the first ideas which comes to mind is to “substract” some known (guessed) clear values from the original message from their encrypted counterparts. For that purpose, it is possible to use the headers and footers of GPG elements. Various substraction need to be tested on the ASCII values of the characters: XOR, substraction with various modulus … After some tests, we quickly get the following:

```
>>> 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 | ######################
```

A visual pattern appaears in the result of this substraction one by one of the characters modulus 26: [1, 19, 19, 22, 5, 20, 9, 7, 0]. To decrypt the message, the idea is to cycle on the table of shift values gathered for alpha characters in the message. Written in 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/OG4yvHmJtHnStoVubGN/Siqgc6C1
4yJOEpiYCGu9j1/IFQDo9GGDzk2GnNRmI+XK9lOhpx6sX2ddfHbfxaJOgCJRHQmd
ssh3uIGXr4pJO4QJ9FCugOT5AM8jVXD2+Rj6k9BVC8C8LORcKhTx9uUGx9Ag2lwD
/R7i4U257WCsZbuF5FqY3qOCl9ELlTJl94qb08+62fJEv5aBepyxeLQcF97kRZ66
L2M1INN4748DWq0vM1UBylOJjpTda1V4OnPeHcUF2ZODmUguYriG5XOFIFBXi38i
eELY2pT3LuIYWwkPh5uVOxa/kVsWaIkidRI5Z3eaL/0nhIwDKpxhBeH2e70BA/0c
YAbu+0TEXqoPhW3qH1Mvbc1Y+jZYCPEHpo69yJHeozuwpOpVeqIixcjMXAb1j3gQ
wJsrXU4YsWn2VX1KLsOVWmWQWlMxXLDhi4jwNmsH2CZbJM72pdWubzezEb8DLfVo
Ja+96C054ipAZSNrS3D1WF5wbO2FBTa0B5Q8I/xg3ck82/+oc0yaoR54eKYF9oUw
uqy7QId+gb35oL4z6SotvOAcMNQj1cCnKNgQiYsnEhTsuv7JZjBqlm16DEzv
=vexd
-----END PGP MESSAGE-----
```

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

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

mQGiBEug+8kRBAC1eP91t96pb/Ja59uVmbz4FuTag7eEFcoWTPUd5mqU9LOOsZH2
N1z5O5p2xeyZmYkp/FcQS8gDCjFMOjCJTxtZFmmeMOo5RbbkWnWXMnc/4PtHerfN
eTeY7mFiVOceMpkonCP7ZMTjA7Ozqq3MorgTu+VvM2iqBir/GSgeHFtsawCg4iUk
8rD+8wmQVJrSFtvLnM/yyi8D/2XCPkXczbj97OCimOgsg/RvdQGnQWcmWF9vLgDZ
KPhlnzwdz2DTwuhfNGjk4SCct06NfE2+PCgufD/eNOQ+jjB5CbvKA64a1WmhUs0w
uKPags6kZ5YZC6T3Mdb2m4jIWaB95+lqGFdOu3Admua2i7NZsT3hDWCWzb6hKZA7
cwHOA/45/FoDLuD4KQtRu5OwUYj5WN3MMVrVxM13bTs3vIEZEZolOHfZI656Sz5P
```

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


>>>
```

A (poor) translation of the text in english gives something like the following:

```
Dear player,

In order to find the treasure, go to the following mysterious
locations:
        - After the rump sessions, let's meet at this breton galliforme
        - The abandoned of Naxos leaves there for new heavens
        - Marvin's brother was born there, under the big table?
        - The nose of this huge captain was only freed in 1993

Once arrived at each location, validate your position in the
application. Add then the password, it is encrypted with GPG.
Then, validate everything and you'll see :)

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

```
hQEOA5BaqIyWyerQEAP9GC73TFXOyby3E49/0G4yvHmJtHnStoVubGN/Siqgc6C1
4yJOEpiYCGu9j1/IFQDo9GGDzk2GnNRmI+XK9l0hpx6sX2ddfHbfxaJOgCJRHQmd
ssh3uIGXr4pJO4QJ9FCugOT5AM8jVXD2+Rj6k9BVC8C8LORcKhTx9uUGx9Ag2lwD
/R7i4U257WCsZbuF5FqY3qOCl9ELlTJl94qb08+62fJEv5aBepyxeLQcF97kRZ66
L2M1INN4748DWq0vM1UByl0JjpTda1V4OnPeHcUF2Z0DmUguYriG5X0FIFBXi38i
eELY2pT3LuIYWwkPh5uVOxa/kVsWaIkidRI5Z3eaL/0nhIwDKpxhBeH2e70BA/0c
YAbu+0TEXqoPhW3qH1Mvbc1Y+jZYCPEHpo69yJHeozuwpOpVeqIixcjMXAb1j3gQ
wJsrXU4YsWn2VX1KLsOVWmWQWlMxXLDhi4jwNmsH2CZbJM72pdWubzezEb8DLfVo
Ja+96C054ipAZSNrS3D1WF5wbO2FBTa0B5Q8I/xg3ck82/+oc0yaoR54eKYF9oUw
```

```
uqy7QId+gb35oL4z6SotvOAcMNQj1cCnKNgQiYsnEhTsuv7JZjBqlm16DEzv
=vexd
-----END PGP MESSAGE-----

Here is my public key if you need to contact me.

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

mQGiBEug+8kRBAC1eP91t96pb/Ja59uVmbz4FuTag7eEFcoWTPUd5mqU9LOOsZH2
N1z5O5p2xeyZmYkp/FcQS8gDCjFMOjCJTxtZFmmeMOo5RbbkWnWXMnc/4PtHerfN
eTeY7mFiVOceMpkonCP7ZMTjA7Ozqq3MorgTu+VvM2iqBir/GSgeHFtsawCg4iUk
8rD+8wmQVJrSFtvLnM/yyi8D/2XCPkXczbj97OCimOgsg/RvdQGnQWcmWF9vLgDZ
KPhlnzwdz2DTwuhfNGjk4SCct06NfE2+PCgufD/eNOQ+jjB5CbvKA64a1WmhUs0w
uKPags6kZ5YZC6T3Mdb2m4jIWaB95+lqGFdOu3Admua2i7NZsT3hDWCWzb6hKZA7
cwHOA/45/FoDLuD4KQtRu5OwUYj5WN3MMVrVxM13bTs3vIEZEZolOHfZI656Sz5P
G73IUF43ZSEi7pkxRLEyLcCpApcQCMWestZGRiOGxgy0FCD0VET1OuzOVn6eAqYz
JRqDnqpcvI4bqUe1G+NPyOgk2BxE+y7mDP5WkSG8eecRlMyWX7QkUGVyc29ubmUg
PGludmFsaWRlQG5vbWRlZG9tYWluZS5ObGQ+iGcEExECACcFAkug+8kCGwMFCQCe
NAAGCwkIBwMCBhUIAgkKCwMWAgECHgECF4AACgkQfh6HQwzuRlDOPgCdHIUXw5wU
ADQBSk1gycl94cCydU0AoImWUlc6t0cufYoYUrh9d/eXq9equQENBEug+8kQBADu
Dv3tRqs7+jDJM2eGjlPg1YMScjTzafGvD3np+rIACphAYhhLH0edD7kM4KNoq39W
YkNIGqW0M4acfA9tVWr9/xcxpjjb7l2GhvBahJVAGSAw4IqrD3u6ea3WrKmkhqIa
FUZq90X5JbGc9Uj1E0ks9U5TsGoI3JSg8FZlmfavQwADBQQAihN4w1JdsLQK0q27
FgSLyG84K6iZ5mP36ZEDqrFFYe6Uxvm8PMziLIalNVJYgR4wKVGeqShQ8zlOvfj0
W7LMtqEZPDZQ8UX/bM89RFSabc/qLYDNVxlbJKfT7mFd/dwB02tpryC/OwEXDEDF
cyzg7IaSvQph4Vcc95xwQOWDfPqITwQYEQIADwUCS6D7yQIbDAUJAJ40AAAKCRB+
HodDDO5GUEA7AKDhvaeXaYoyjLLftlQY4WDssi91vgCfWWNioOoCfm0eTgCNc/im
ZBJoifI=
=8IMk
-----END PGP PUBLIC KEY BLOCK-----


>>>
```

The decrypted version of the file is also embedded in the document for the interested reader.

The rest of the challenge involves providing Secret application the GPS coordinates of 4 locations (the ones described in the riddles) and a password (to be extracted from the two GPG elements found in the message).

6 Answers to the 4 questions

6.1 After the rump sessions, let's meet at this breton galliforme

This year like previous year, SSTIC social event takes place at the Coq[7] Gabdy, 156 rue d'Antrain, in Rennes, France. The first GPS coordinates are offered as a gift:

- Latitude : 48.123
- Longitude: -1.667

6.2 The abandoned of Naxos leaves there for new heavens

According to Wikipedia, Naxos is a greek island in the Aegean sea. It is famous partly due to the mythology: according to the legend, Theseus abandoned Ariadne there . . . . The launch pad of Ariadne rocket in Kourou in Guayana is located at the following GPS coordinates:

- Latitude : 5.15865
- Longitude: -52.650261

6.3 Marvin's brother was born there, under the big table?

The answer to this question was not found by the author. Valid coordinates have nonetheless being obtained by reversing libhello-jni.so library:

- Latitude : 37.88
- Longitude: -122.3

The answer to the question is a location near San Francisco.

6.4 The nose of this huge captain was only freed in 1993

The Nose is one of the original climbing routes up El Capitan in Yosemite valley in the USA. It was first freed by Lynn Hill in 1993. Associated GPS coordinates are:

- Latitude: 37.734
- Longitude: -119.63737.73

[7]cockerel, rooster in english

6.5 Note

After the reverse engineering of libhello-jni.so described later in the document, the following information on coordinates are available:

- The eight components of the coordinates (4 pairs of latitudes and longitudes) are initially multiplied by π before being passed to the deriverclef() key derivation function from libhello-jni.so.
- The 8 coordinates components are rounded and then converted to integers. The final result is an 8 bytes table.
- The 8 bytes table is then used to reconstruct the name of a class which is then used later in the program. An invalid value for one of the component prevent access to this class.
- The 4 components (associated with the two easy questions) are used in the key derivation procedure. The last 4 ones are not used in the key derivation procedure.

More details are given in the section discussing the reverse of the library.

7 Recovery of GPG-encrypted password

7.1 Introduction

The password asked to the user by Secret application after passing GPS coordinates is GPG-encrypted. We first start by analyzing the provided GPG elements.

A GPG element (public key, private key, encrypted message, . . .) is a data stream which can be split in a set of data packets. Each packet has a tag associated with its type, a length and data (specific to its type). Each GPG element has a specific structure made of a set of those packets. The format of those GPG elements and of packets is documented in RFC 4880.

A GPG element is basically a binary blob. An ASCII encoding (ASCII Armor) of this binary blob is defined: the GPG element is base64-encoded, it is concatenated a (3 bytes) checksum computed before the base64-encoding and is then prepended a header and a trailer. This is the format of the two GPG elements found in the message.

7.2 Analysis of GPG public key

The GPG ley has been saved in a file **pubkey.pgp**. We start by analyzing it in detailed fashion.

The --list-packets option **gnupg** provides some initial details on the key. The numerical values (algorithms, functionalities, ...) returned by the following command are documented in RFC 4880.

```
$ 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]
```

The first packet matches the public key: a 1024 bits DSA key. The second packet contains the identity of the key owner (User ID): "Personne <invalide@nomdedomaine.tld>".

The next packet is the signature of the ID and previous public key (with associated private key). The signature packet contains preference information, in particular about cryptographic and compression algorithms which can be used during the exchange of data using GPG. In particular:

- **Symmetric encryption algorithms**: AES256 (9), AES192 (8), AES128 (7), CAST5 (3), 3DES-EDE (2)
- **Hash algorithms**: SHA256 (8), SAH-1 (2), SHA384 (9), SHA512 (10), SHA224 (11)
- **Compression algorithms**: ZLIB (2), ZIP(1)

The main key of the user being a DSA key, it is not usable for operations others than signature. The next packet contains a 1024 bits ElGamal subkey. It is signed by the the DSA key of the user (last packet). The elements noted `pkey[0]`, `pkey[1]` and `pkey[2]` correspond respectively to prime number p (1024 bits), groupe generator g (3 bits) and public value $y = g^x \ mod \ p$ (with x the secret part).

The extraction of the 3 elements of the ElGamal public key is pretty straightforward with the information from RFC 4880 (by hand) or using **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 of GPG-encrypted message

The GPG-encrypted message contains the following elements:

```
$ 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
```

The data of the message are available in last packet, encrypted using a symmetric encryption algorithm (available only to the target of the message), via a session key generated specifically for that message. The first packet of the message contains that session key encrypted using the public part of Personne's ElGamal key. The second packet contains this same session key encrypted using RSA. The specific owner (ID 0x2A9C6105E1F67BBD) of this key is unknown.

The two 1021 bits elements (the ElGamal-encrypted data) in the first packet of the message are respectively $c_1 = g^k \ mod \ p$ andt $c_2 = m \cdot y^k \ mod \ p$ with:

- g the group generator (i.e. 5) given above,
- y the ElGamal public part of Personne
- p the modulus
- k the ephemeral ElGamal key used for encrypting the session key.

This parameters which we will use later can once again be extracted by hand or using **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 Unsuccessful leads

Before describing the definitive solution (a quite logical one in the end), quite some time has been spent (lost) by the author on many unsuccessful approaches.

The search for the private part of ElGamal key in the memory dump did not give any result.

The search for the session key used to encrypt the data of the message in the dup did not give any result (for AES128 and AES256).

This two paths gave no result so it seemed logical to thing the block containing the RSA-encrypted session key has some interest for the challenge. In the end, the author was unable to extract anything useful from it for the purpose of the challenge.

To justify that work, it just **initially** seemed too simple, obvious and not that fun after the decryption of **chiffre.txt** to test the parameters of the key.

7.5 A solution

7.5.1 Analysis of ElGamal parameters

In the encryption of the message using GPG, ElGamal has been used to protect a session key associated with a symmetric algorithm to protect the data itself. Preference of user's key suggest AES256 may have been used but the confirmation will only be available after the session key has been decrypted.

The first test to perform on public parameters of ElGamal key against primality of the modulus, p.

```
>>> from numbthy import *
>>> p = 1671704073483687552745711987108461550058521974511 5\
    830499928225759380624969023546140058947540538438154345\
    145620177131155416701308682048739416793020860201506654\
    676963823542017623213731147411563999950320483744863834\
    614329486886495306347325615024815752435650939454556953\
    1012643391645942093695428659549471312883523
>>> isprime(p)
True
```

p being prime (as expected), the order of the group is $n = p - 1$. Let's check if p is a safe prime by verifying that $n/2$ is also prime.

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

Weird. Let's try and factor n. To do that, **PARI/GP** is a more suitable (i.e. efficient) tool than Python.

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

? factorint(1671704073483687552745711987108461550058521974511583049\
            9928225759380624969023546140058947540538438154345145620\
            1771311554167013086820487394167930208602015066546769638\
            2354201762321373114741156399995032048374486383461432948\
            6886495306347325615024815752435650939454556953101264339\
            1645942093695428659549471312883522)
%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]

?
```

The output is made of the list of prime factors of n with their associated exponent in the decomposition. Here, each factor only appear once in the decomposition, .i.e. the prime factors all have an exponent of 1.

factorint() uses a combination of methods for factoring (see PARI/GP manual for details). The primality of the value returned by PARI/GP is not guaranteed but can easily be tested using **isprime()**. This is the case of returned values.

All returned values all have similar sizes (51 bits) except for the last one, which is a bit larger (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
```

The existence of such a decompostion of the order of the group in prime factors of reasonable size makes it possible to use Pohlig-Hellman algorithm to compute discrete logarithm:

- either to recover the value of x, the Elgamal private key, i.e. find x such as $y = g^x \ mod \ p$
- either to recover the value of k, the ElGamal ephemeral key i.e. find k such as $c_1 = g^k \ mod \ p$

Recovering either value is sufficient to recover m.

Pohlig-Hellman algorithm solves the discrete logarithm in Z/pZ by computations of discrete logarithms in subgroups with smaller orders (with sizes associated to previous primes) followed by the resolution of a congruence system using the Chinese Remainder Theorem. The computation of discrete logarithm in the subgroups can be performed using Pollard Rho algorithm.

The interested reader can find a good description of Pohlig-Hellman algorithm in section 3.6.6 of chapter 3 of Handbook of Applied Cryptography by Menezes, van Oorschot and Vandstone (freely available for download).

In our case, the complexity of the attack is bound to the computation of the discrete logarithm in the subgroup associated with the largest factor. An average complexity is $O(\sqrt{3165493139633045911}) = O(2^{31})$.

Next section describes the setup of the attack, i.e. the tools used to recover one of the two discussed values.

7.5.2 Changes to Pohllig-Hellman implementation found in LiDIA

To perform the computation, we use a modified version of Pohlig-Hellman implementation (**dlp_appl**) available in LiDIA.

The tool **dlp_appl** available in the examples of the library takes as parameters a, b et p and returns x such as $b = a^x \ mod \ p$. To do that, it first tries and factor $p - 1$ on its own. unfortunately, the program has a limitation on the size of numbers it tries and factor. And p is larger than that limitation ...

Because we already have access to the factors of $p - 1$ (using **PARI/GP**), the first step is to quickly modify the program to make it accept the factors as input parameters. The patch associated (**use_factors.patch**) is embedded in the 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
```

We can then start the program by passing it a and b directly followed by the list of prime factors of $p-1$. In our case:

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

will return x such as $b = a^x \; mod \; (p1 \cdot p2 \cdot \; \ldots \; \cdot p20 \cdot p21 + 1)$

7.5.3 Obtaining x et/ou k

As described previously, obtaining either x or k is enough to recover the message. Nonetheless, the complexity of the algorithm is an upper bound. The computation time will depend from x and k: one of the discrete logarithm computations will end before the other.

In practice, the two computations have been run in parallel on the same box (Intel Xeon running at 3GHz). The computation of k completed after around 7 hours. The computation of x completed after around 21 hours.

For the computation of k, the intermediate results of Pohlig Hellman algorithm are provided below:

- $1 \; mod \; 2$
- $872369050350763 \; mod \; 1218055055968339$
- $1068041787979718 \; mod \; 1263847861201609$
- $958955728833198 \; mod \; 1271483404519507$
- $408764087210986 \; mod \; 1306620742471661$
- $571880503612888 \; mod \; 1435469233657999$
- $1151793512882267 \; mod \; 1436852757281407$
- $715557480710616 \; mod \; 1455144603998677$
- $1064806456619047 \; mod \; 1593684693149279$
- $1244667582116960 \; mod \; 1724498562415303$
- $1261679303781737 \; mod \; 1780716924867173$

- 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

The final result is:

```
k = 3442798784842268266210142027076483000991667721152362526627\
    90990317493690573
```

For the computation of x, the intermediate results of Pohlig Hellman algorithm are provided below:

- 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

- $868725239079599 \ mod \ 1922550339910303$
- $243798483328639 \ mod \ 1975985172968039$
- $1749977111909658 \ mod \ 2077649398994551$
- $1258511240243730 \ mod \ 2108107767794563$
- $1851179581442108 \ mod \ 2132773087614569$
- $1648685792733093 \ mod \ 2133463604190461$
- $738043239287262 \ mod \ 2174110522001753$
- $923697751917423 \ mod \ 2227343475745711$
- $1458660248217431679 \ mod \ 3165493139633045911$

The final result is:

```
x = 8216000284630378878087764573899538969348205656695755958036\
    9371661088239455998708979230174270492184887093527756473793\
    0378467851629800295372362462697283479462781737323943230799\
    06059057656454597026161020547O7
```

7.5.4 Password recovery

k being the first result obtained, we use it to recover the message. We first start by checking the result of previous computation is correct:

```
>>> from numbthy import *
>>> g = 5
>>> p = 1671704073483687552745711987108461550058521974511583049992822257\
        5938062496902354614005894754053843815434514562017713115541670013\
        0868204873941679302086020150665467696382354201762321373114741115\
        6399995032048374486383461432948688649530634732561502481575243356\
        509394545569531012643391645942093695428659549471312883523L
>>> k = 3442798784842268266210142027076483000991667721152362526627909990\
        317493690573L
>>> c1 = powmod(g, k, p)
>>> c1
1698220381424689846971375433580431781140043898932757760264726551247178\
9478232528615066217063121173149615046780018928268217187606793500122178\
0540964587915262352200461846645018469753793985668882594271768475258274\
9539690346478984977420723994860330962159118795043400607203277255531780\
0569333555636242534139880L
```

It is indeed c_1. Now:

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

We have

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

Which can be written in the following way in Python (modinverse module is embedded in this document):

```
>>> from Crypto.PublicKey.RSA import number  # for long to bytes conversion
>>> from modinverse import modinv
>>> c2 = 21689062591603567787867095122856828505852579982431459845376876\
         58000615088122707550661679953960668083472453023489055647422779\
         45265292217455204099111023604397954835919856674937343744172714\
         01161725037353235503230119784387210884625936483054829095607196\
         0622349914955749362379391540259341985931041881363939537707791L
>>> y = 96960307715502623841883822033776180874428468574363119048730370 4\
        83858304212727958408600652044040532527114718847920067959183307 5\
        31117366537777881085966200583722563728398158892468986104185332 2\
        14967529194102414162432879696273143394583000301199628705089184 9\
        5298713435327116381934538095456793047979087444803659289 0L
>>> k = 34427987848422682662101420270764830009916677211523625266279099 0\
        317493690573L
>>> p = 16717040734836875527457119871084615500585219745115830499928225 7\
        59380624969023546140058947540538438154345145620177131155416701 3\
        08682048739416793020860201506654676963823542017623213731147411 5\
        63999950320483744863834614329486886495306347325615024815752435 6\
        5093945455695310126433916459420936954286595494713128835 23L
>>> m = c2 * modinv( pow(y, k, p), p) % p
>>> m
67546661681506969929693066680890881415578055696473749545605776073108061 4\
03696088731254939462263920481828033384665385560338455378796028310209290\
88661331835747714458601506848585737141685949321238591887996079923468829\
15320532409373802269161061459134784995523019803068494519168146040329042\
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'
```

Joy! The message has the expected format, i.e. a type 2 PKCS#1 block as described in section 7.2.1 of RFC 3447:

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

With:

- EM (Encoded Message): encrypted bloc
- PS (Padding String): made of non-zero random bytes

- M (Message): the message

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

As described in RFC 4880, the encrypted message is not built only with the symmetric key: it is followed by a 16 bits checksum and prepended with one byte indicating the symmetric encryption algorithm used:

```
>>> 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
```

The algorithm used for encryption is AES256 which matches the 32 bytes of the key. AES mode used by GPG for encrypting messages is specific: it is a variant of CFB mode.

It is worth mentionning that support for that mode is announced in Py-Crypto but simply does not work. It is possible to implement it quite quickly. Once done:

```
>>> 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()` has not returned None, so the decryption went ok (check of the two duplicated bytes as described in section 13.9 of RFC 4880). The first byte of the result indicates the kind of GPG paquet: a type 8 packet, i.e. compressed. The following byte gives the kind of compression used: 2 for zlib.

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

The filename (mdp.txt) and its content (O7huQcYzHEPSq82m) are in a Literal Data packet (Tag 11 in the first octet) which format is described in section 5.9 of RFC 4880. The next octet indicates the length of the packet: 0x1e, i.e. 30 octets.

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

The next byte (`d[1]`) indicates the type of data: 'b' for binary data. It is followed with the filename prepended with its length (mdp.txt' of length 7), and then with a date (November 25 2010 14:24:28) associated with the file:

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

The content of the file (i.e. the password) follows:

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

>>>
```

And the password is then: **O7huQcYzHEPSq82m**

7.5.5 Conclusion

The specific form of the modulus associated with the ElGamal key of Personne allowed us to quite easily access its secret part and then the ephemeral key used to encrypt the message.

This then provided the session key used to encrypt the message and then the message itself.

This part also gave us the opportunity to spend some time on GPG messages format.

Nonetheless, a question remains: what was the purpose of the RSA block in the encrypted message?

8 Analysis of classes.dex from com.anssi.secret

In order to understand how Secret application precisely works, it makes sense to analyze its main file, **classes.dex**.

It seems pointless to try and analyze the content of .dex file directly. A first disassembly step is required to convert this file to a higher level language.

For that purpose, we use **baksmali** (islandic equivalent of "disassembler") from smali project. The latest version (1.2.2) of the application (a JAR) and its launcher (a script) can be downloaded:

```
$ 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 can directly process .apk files and generate a set of .smali files containing the implementation of the various classes found in the 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
```

With some documentation, the high level assembly language found in .smali files is easily readable. A first pass on the seven resulting files provides two interesting leads to follow: SecretJNI.smali et RC4.smali. For the interested reader, the 2 files are embedded in the document. Their content is detailed below.

8.1 SecretJNI.smali

SecretJNI.smali contains following methods:

```
$ 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
```

The most interesting methods are discussed below:

8.1.1 clinit()

clinit() constructor contains 3 interesting elements:

- A variable 'coincoin' containing a string which seems to be base64-encoded. No reference to this string appears in the rest of the application.

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

  This sentence is a private joke and is just useless in the resolution process.

- A 'programme' variable containing a 2548 octets hexa-encoded string. This is used only in the dechiffrer() method: it is converted to a byte array (via a call to hexStringToByteArray()), the result being then passed to the crypt() method of an instance of RC4 class (initialized with a key passed as parameter to dechiffrer()).

  The string 'programme' (newlines have been added)

  ```
  "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"
```

- A call to `System->LoadLibrary(‘‘hello-jni’’)`

8.1.2 dechiffrer()

As discussed above, the method takes a string “clef” as parameter. It accesses ‘programme’ instance variable (hexa string discussed above), which it converts to a byteArray. The byteArray is then decrypted (via the RC4 implementation) with a key passed as parameters to the method. Decrypted string is returned.

8.1.3 deriverclef()

The implementation of this virtual method is provided by libhello-jni.so library loaded by clinit(). It is called from onClick() and its output is passed to dechiffrer() method. It takes as parameter a reference to the JNI environment, an array of location (GPS coordinates) and a string (password).

8.1.4 onClick()

In spite of its name, this method is the heart of the application. The analysis of its behavior is pretty straightforward. It works in the following way.

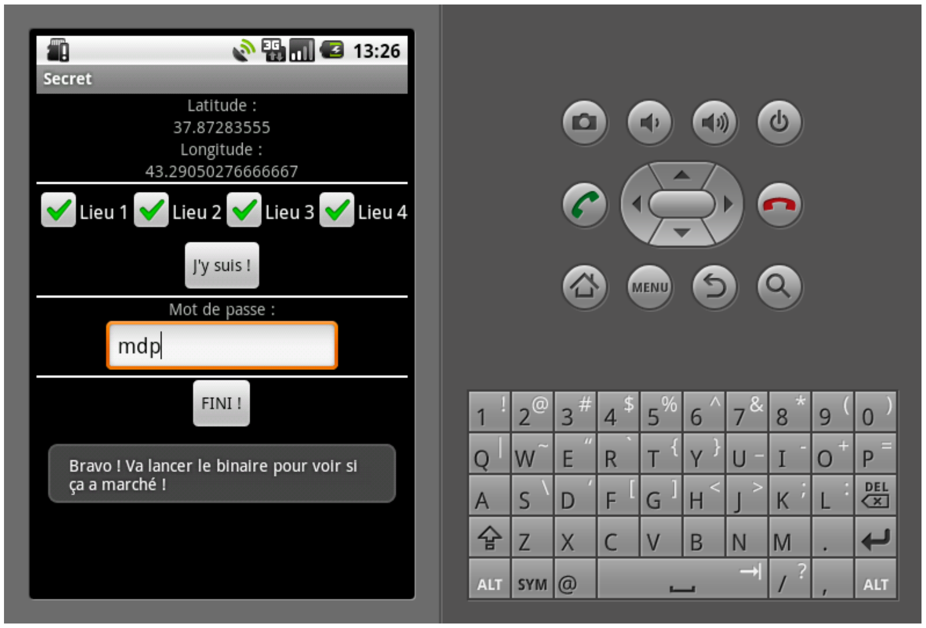
It first starts by gathering GPS coordinates (latitude and longitude) of the four locations which are then stored (after multiplication by π) in a table of double. The user is then offered the choice to enter and then validate the password.

At that time, the table of locations and the password are passed to deriverclef() virtual method (which implementation is provided by libhello-jni.so). **A**

priori, it returns a key which is then passed to dechiffrer() method described above.

For practical purposes, invalid geographical coordinates passed to the application make it crash. The reasons for thart crash are detailed in the section presenting the reverse engineering of libhello-jni.so library.

The decrypted string returned by the function is saved on disk in the application folder as "binaire"[8]. A message ("Bravo ! Va lancer le binaire pour voir si ça a marché !"[9]) is displayed as shown below:



Latitude :
37.87283555
Longitude :
43.29050276666667
Mot de passe :
mdp
Bravo ! Va lancer le binaire pour voir si ça a marché !

Figure 9: Result of passing good coordinates (no matter the password)

The password being used to generate the decryption key, it must be valid: otherwise, the file saved on disk is just garbage:

```
$ ./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 ")")
```

Obviously, the result is somewhat different with the valid coordinates and password:

[8] french for "binary"

[9] "Congratulations ! Go and launch the binary to see if it worked!"

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

In the end, except if the answers to the four questions and the password have been recovered, here are the next steps

- study the processing of input parameters (coordinates and locations) by deriverclef(), in order to try and understand the derivation logic, and possibly infer valid values for this input paramaters. This understanding requires reversing libhello-jni.so library. This is described in next section.
- study RC4 module used for decryption. This analysis is presented below.

8.2 RC4.smali

RC4.smali file provides **a priori** the implementation of a module for the RC4 encryption algorithm. It is used directly by SecretJNI.smali but also by libhello-jni.so. The methods of the module (detailed below) are the following:

```
$ 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
```

When studying libhello-jni.so, it appears that com() static method is used. It takes as parameters a key and a table of data to encrypt. The study via gdb of input (keys and string) and output (encrypted input string) values, and the comparison of those with the result obtained with an implementation of RC4 shows differences: either the module does not implement RC4 in spite of its name, or it is a slightly modified version of RC4.

Even if it would be possible to consider the module in an opaque fashion and reuse it for our computation, it is small enough (300 lines of smali) to be completely studied:

- **public constructor <init>([B)V**: init() constructor takes as parameter a key passed as a table of bytes. It is used to initialize the internal state (a table of 256 bytes) of RC4 instance.
- **static com([B[B)V**: this static method of the class take as parameter a key and data (both passed as tables of bytes). The key allows initializing an instance of the encryption algorithm and then encrypt data via a call to cryptself(). The table of data passed as parameter is modified for output.

- **public crypt([B)[B**: this method is used to encrypt data passed as argument (as a table of bytes). It returns a table of bytes corresponding to the encrypted data.
- **public cryptself([B)V**: this method is similar to previous crypt() method but encrypt data passed "in place" and does not return anything.
- **getbyte()B**: it is an internal method used by crypt() and cryptself(). It allows accessing to the keystream bytes (and performs the required modifications to the internal state of the algorithm).

An analysis of the getbyte() method against a reference implementation does not show any specific difference. Nonetheless, the init() constructor, even if it performs a standard initialization from the key, also contains an additional step: it drops the first 0xc00 (3072) first bytes of keystream. Along with the developers, it seemed unreasonable to name the module RC4DROP3072 ...

In the end, an encryption/decryption function similar to the one in the module is obtained with the following lines of Python:

```python
from Crypto.Cipher import ARC4

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

9 Reverse of libhello-jni.so

9.1 Introduction

As discussed previously, the key derivation from the coordinates and password is performed by the virtual method deriverclef(), whose implementation is provided by libhello-jni.so library.

This section gives details on the reverse performed on this library to understand how this derivation is perfomed. The purpose of this operation is also to obtain information on coordinates and password, and possibly recover those (directly or via brute-force).

The reverse of the library has been performed "by hand" using only **objdump** and a text editor (**emacs**). **vi** would also have done the job.

9.2 First pass on objdump's output

libhello-jni.so library is obtained simply by unzipping the APK of Secret application.

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

In order to disassemble the library, we use a version of **objdump** supporting ARM architecture. This kind of tool is usually not directly available in Linux distributions but is also simple to install:

- Debian Embedded project provides directly installable versions of the tool for various architecture, including ARM.
- The previously installed Android NDK also provides a version (build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/arm-eabi-objdump).
- The owner of ARM devices may also install and use a native version of the tool. **objdump** is for instance available on Nokia N900.

The last option has been used by the author but the second is probably the simplest one for most users considering the needs:

```
$ 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
```

In a text editor, we can start then start the analysis of the library. A first pass on its content indicates that:

- function names in the library are all obfuscated except deriverclef().
- parts of the library, in Thumb mode (16 bits instructions), do not have been disassembled correctly. This is not a problem at the moment. It will be possible to disassemble those parts of the library using `--disassembler-options=force-thumb`.
- data elements have been disassembled as code, for instance at the end of 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]!
  ```

- deriverclef() function has 250 lines of assembly and calls various subfunctions (with obfuscated names: G4Cy, SXXJZ, QUZd7pH7J, . . .).

The code associated with the beginning of the function is presented below:

```
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]
...
```

The next step simply consists in using a good documentation on ARM assembly[10] and "read" assembly code to understand the actions performed from the GPS coordinates and password passed as parameters to the function.

To be short, the rest of this section makes the hypothesis that the reader is somewhat familiar with ARM architecture.

It is sometimes good to check the conclusion associated with the reading of the code by studying the state of the register of the processor and the state of the application's stack using GDB. GDB installation has been detailed previously.

Presenting here a line by line explanation of the function is not that interesting. Instead, the structure is detailed below (with some code extracts) with the difficulties encountered.

9.3 Structure of deriverclef()

The first thing to note on the application is that it allocates a large amount of space on the stack at address 0x1734:

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

We will see that this space is used:

- for local variables,
- for computations performed by a hash algorithm,
- to reconstruct (from the GPS coordinates and elements available in rodata section of the library) the name of a class.
- for converting GPS coordinates
- ...

After saving registers r4 to r12 and previous allocation on the stack, the function performs a call (at 0x174c) to **const char* (*GetStringUTFChars)(JNIEnv*, jstring, jboolean*)** method from JNI environment to convert the password in a simple string. The computation preceding this call are initially a bit puzzling but deserve to be explained because the mechanism is used multiple times later in the code:

[10]For example, http://infocenter.arm.com/help/index.jsp

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

Previous code computes a fixed shift value ($169 \ll 2 = 169 * 4$) in JNINativeInterface structure (to which r2 points). The file defining the structure is available in Android NDK:

```
$ 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);
...
```

In the end, the computation provides access to the 169^{th} element of the structure, i.e.

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

After this call, 17 bytes from rodata are copied on the stack (starting at sp+388). Those will be used later in conjunction with the GPS coordinates to construct a class name. Three bytes after the end of copied data (at sp+408), 8 bytes of rodata section are copied again: those will serve later as signature to call a static method of previous class.

Let's use GDB to access the content of the stack at sp+388 after the copy by placing a breakpoint, for instance at 0x1780. Once the application launched in the emulator, **ps** provides access to its PID: 298 on our example. By studying the content of cat **/proc/298/maps**, we get the address at which the library has been mapped (fixed from one launch to another).

```
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
```

Outside the emulator:

```
$ ./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                  # required for some
Breakpoint 2 at 0x80a01781           # unknown reason
(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
```

The bytes are indeed those copied from rodata section. Their use is discussed a bit later.

Then, the code from deriverclef() call a function in the library, with an obfuscated name (<G4Cy>): the parameters passed (using r0 et r1) to this function are a pointer to a static buffer in the stack (at sp+28) and the value 256. A quick analysis of the code of this function and additional hints obtained from the reading of deriverclef() can let us think it is an initialization function for a hash algorithm:

```
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 with 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 with 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 with 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 with 192
13fc: d001    beq.n 1402 <G4Cy+0x3a>
13fe: 29e0    cmp r1, #224            # comparaison with 224
1400: d1f9    bne.n 13f6 <G4Cy+0x2e>
1402: 4651    mov r1, sl
```

The values against which the second parameter is compared can **initially** let us think to SHA2 algorithms. This hypothesis will later be invalidated.

In the remaining of the code, a 32 elements jbytearray is allocated (at 0x1796) via a call to the 176th function of JNIEnv:

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

It will later be used to store the output of the hash algorithm called on the password in order to be able to pass it to a function of RC4 module.

Once the allocation has been performed, the code of derivercIef() computes (between 0179c et 17a6) a NOT of each of the 8 bytes from sp+408. Those are the last 8 bytes obtained previously with gdb. A check of the same memory area after this loop validates it:

```
(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
```

The continuation of deriverclef() becomes more interesting, because it processes the GPS coordinates passed as parameter to the function. A call to the 190th function of JNIEnv

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

provides access one by one to the elements of jdoubleArray containing the GPS coordinates as jdouble (i.e. double, i.e. 64-bit IEEE 754). Each of those is then passed succesively to 2 unknown functions from the library:

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

An analysis of the code of those 2 functions indicates a large number of mathematical computations on the inputs and the calls to some subfunctions also available in the module. Before starting "by hand" a detailed analysis of the function, it seems judicious to check those are not simply functions of common library (libc, libm, . . .) statically integrated to libhello-jni.so. For that purpose, it is enough to disassemble those common libraries with **objdump** and look for a specific pattern. For instance, the `pop {r4, r5, r6, r7, pc}` in the middle of the first function is a good candidate:

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

It is quickly found in disassembled code of libm, in the middle of round() function (libm available on the system is obviously not obfuscated). An quick analysis indicates that the code of the two functions is identical:

```
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
...
```

The second function (QUZd7pH7J) performs a conversion from double to integer. Its code is nearly identical to the one of `__aeabi_d2iz()` found in libc.

In the end, the loop between 0x17c2 et 0x17de transforms (via rounding and conversion to integer) the table of GPS coordinates to a simple 8 bytes table. For each coordinate value passed by the user, the result in the 8 bytes table is the output of the following Python snippet:

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

The 8 bytes table is stored on the stack starting at sp+416.

Then, derivercleff() calls strlen()[11] to get the length of the password.

The elements below are then passed to **8j3zIX** function:

- a pointer to the data area initialized in stack at sp+28[12].
- a pointer to the password
- the length of the password returned by **strlen()**
- 0

It is then called once again, this time with the following parameters:

[11]the call is performed via the plt and the got and must be resolved by hand
[12]by the function we suspect to be an initialization function for a hash algorithm

- the same pointer to the data area initialized in stack at sp+28
- a pointer to a 8 bytes table containing the converted coordinates
- 32
- 0

An important point to note here is about the 3^{rd} parameter. During the first call to the function, the password is passed with its size in bytes. For the 8 bytes table, the value 32 is passed as length. An analysis of the beginning of **8j3zIX** function can let us think that it considers its third parameter as the length of its second in bits. If our hypothesis are correct, it could be an update function of the hash algorithm.

After this 2 calls, deriverclef() performs a call to **sdlHj** with the following parameters: the data area in stack at sp+28 and a pointer to sp+356. The second parameter seems to be used as a static output buffer:

- data at sp+388 can let us think it is a 32 bytes buffer.
- the following function called by deriverclef() is SetByteArrayRegion() from JNIEnv which performs a copy of the 32 octets at sp+356 to the jbytearray allocated at the beginning of the function.

If our hypothesis is correct, it could be the initialization function of the hash algorithm. The 32 bytes output (i.e. 256 bits) corresponds to the value passed (in bits) to the initialization function. The rest of the story on the supposed hash algorithm contiues in the next subsection.

The continuation of deriverclef() (in 0x1828) performs a XOR of the 17 bytes previously copied from the rodata section of the library (and then NOTed) at sp+288 with the 8 bytes of coordinates (repeated). The result is progressively placed at sp+288. The few following lines of the function complete the work by taking the 4 first bytes placed at sp+288 and by XORing them with 3 fixed values from the function (49, 44, 89, 47) to then add them after the first 17:

```
1848: 7822   ldrb r2, [r4, #0]  // r4 points 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]
```

If provided GPS coordinates are invalid, the resulting string is invalid. The questions to the first two questions being offered, we initially get a partially valid string like the following:

```
$ 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"
```

This string being then passed to the FindClass() method of JNIEnv, it is necessary to make it valid by finding valid GPS coordinates. The implementation of RC4 being part of the application, the expected string seems quite obvious: **com/anssi/secret/RC4**. Fincding the last 4 bytes of the locations table and then recover the value to pass to the application is immediate.

The set of valid (lat, long) values is:

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

It is easy to verify with gdb that the name of the class is correct in stack with this values:

```
$ 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"
```

The continuation of derivercllef() is quite simple. After the call to FindClass(), a call to GetStaticMethod() to access a static method is performed, followed by a call to CallStaticVoidMethod() to call this method. The only static method of RC4 module is com() method. The output of the supposed hash algorithm (32 bytes previously placed in the jbytearray) is passed as key but also as data. The result will then be formated (as an hex string) to be returned by deriverclef() to the caller.

Next subsection discusses the expected hash function. The subsection following it terminates the recovery of the password based on the information obtained in current subsection.

9.4 Supposed hash function

To confirm our hypothesis on the use of a hash function in deriverclef(), we start by analysing what is in the stack between sp+28 and sp+288. **gdb** is our friend:

```
$ 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
```

Instead of comparing the values of the table with initialization values of existing hash function for different sizes of hash, a more efficient and simple method is prefered: let Google do the job.

A search for the 8 octets (``b405f031 55cb34a5'') found in the middle of the buffer (starting from 0xbed5581c above) gives only this 2 response, for the same hash algorithm: Shabal.

Shabal is a hash function submitted by SAPHIR research project to NIST international competition to find a new hash function (SHA-3). DCSSI (former name of the ANSSI) is among the contributors and the function supports hash sizes of 192, 224, 256, 384 and 512 bits.

Shabal submission document (a 300 pages PDF) contains in its appendix A a reference implementation. The init and update function accepts length in bits.

An extraction of this reference implementation and the addition of a main() is useful to perform some tests to verify it is indeed Shabal or a modified version.

The resulting code is embedded in this document (shabal.c, shabal.h). The main() is given and commented below:

```
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");
}
```

The GPS coordinates are static (valid). The first four bytes of the password (the decimal values of associated[13]) can be passed as arguments. The 5th argument of the program is the length of the password in bits, passed directly to updated function. No sanity check is performed on the input parameters.

This implementation can be used to verify previous hypothesis i.e. that the size of the password is passed in bytes whereas the update function expect a value in 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
```

The test are done by passing to Secret the GPS coordinates GPS available in the source and the four passwords starting by “toto”: “toto”, “totototo”, “tototot” et “tototototototo”.

For “toto”, we get the result below:

```
$ 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
```

[13] to simplify later instrumentation of the function

```
...
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
```

We get the result obtained with our first call to our test program above (`./shabal 116 111 116 111 4` gave us `78e765d31...`). We perform the same check for the 3 others values. Respectively, for "totototo", "totototot" and "totototototototo" passed to the application, we get:

```
(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
```

Our tests confirm all our hypothesis:

- The hash algorithm is indeed Shabal Shabal (256)
- The size passed to the update function for the password is the length of this password in bytes whereas the function expects bits.

The conclusion is that the entropy contributed by the password to the derivation function is extremely reduced: 8 bits for a password between 1 and 7 characters, 16 bits for a password between 8 and 15 characters, ...

We use that fact in next section to perform a brute-force to recover a valid password and the binary.

10 Getting of a password and the binary

Before discussing the recovery of the password expected by Secret and the associated binary, let's make a summary of the results obtained in previous section:

- The encrypted binary is available (directly in SecretJNI.smali)
- The encryption algorithm used is RC4DROP3072
- The key is obtained by a call to deriverclef()
- The key returned by deriverclef() is a hash encrypted using RC4DROP3072. The key is the hash itself.
- The hash algorithm used by the key derivation function is Shabal (256)
- Hashed data are the 8 bytes of coordinates and the N first characters of the password ($N = len(mdp)/8$)
- The 8 bytes of coordinates have already been recovered

The only remaining step is a simple brute-force on the password to test if derived key decrypts a valid binary.

This is written in few lines of Python using simple calls to previous **shabal** binary. This is not efficient at all but the domain to scour in search of a password of at most 23 characters is very limited ($1 + 256 + 256^2 < 66000$). The following code tests all the passwords between 16 and 23 characters.

The stop condition (hypothesis confirmed afterwards) is that expected binary is in ELF format. **shabal** binary must be available in current folder.

```
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 "[+] Binairy found! "
                print "[+] Beginning of password '%c%c'" % (chr(i), chr(j))
                open('binaire', 'w').write(d)
                return
```

Une fois lancé, on obtient le résultat

```
>>> bruteforce()
[+] Binary found!
[+] Beginning of password 'O7'
```

The first 2 characters of the password recovered by decrypting the GPG message (O7huQcYzHEPSq82m) match those returned here.

The binary is available incurrent folder under the name "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
```

Let's just push the binary in the emulator (or enter GPS coordinates and a password starting with O7 having a length of 16 to 23 characters in Secret)).

```
# 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

In the end, the two possible paths to solve the challenge discussed in the document – the first one more crypto-oriented and the second more reverse-oriented – have both permitted to spend time on very interesting technical aspects.

Solutions proposed in the document are probably neither the only existing ones nor the most efficient but they have allowed us to discuss the various technical domains covered by the challenge.