InsomniDroid CrackMe Spoiler
Insomni'hack 2012

Axelle Apvrille, Fortinet

March 2012

Abstract

This is the solution to the "InsomniDroid" challenge which was released at Insomni'Hack [Ins12] challenge, in Geneva, March 2nd 2012.
If you are willing to try the challenge, stop reading this document as it spoils it all!

1 Challenge

The challenge consists in a single APK file, with no particular additional explanation.

```
-rw-r--r--  1 axelle axelle 15076 Mar 19 11:47 insomnidroid.apk
```

2 First glance

The .apk extension is for Android Packages, and it looks like this package is indeed an Android package: it uses the ZIP format and note the presence of the typical classes.dex (Dalvik Executable) file.

```
$ file insomnidroid.apk
insomnidroid.apk: Zip archive data, at least v2.0 to extract
$ unzip -l insomnidroid.apk
Archive:  insomnidroid.apk
  Length      Date    Time    Name
---------  ---------- -----   ----
     1120  2012-02-16 09:35   res/layout/main.xml
     1336  2012-02-16 09:35   AndroidManifest.xml
     1900  2012-02-16 09:35   resources.arsc
     3966  2012-01-24 16:37   res/drawable-hdpi/icon.png
     1537  2012-01-24 16:37   res/drawable-ldpi/icon.png
     2200  2012-01-24 16:37   res/drawable-mdpi/icon.png
     3656  2012-02-16 09:35   classes.dex
      564  2012-02-16 09:35   META-INF/MANIFEST.MF
      617  2012-02-16 09:35   META-INF/CERT.SF
      842  2012-02-16 09:35   META-INF/CERT.RSA
---------                     -------
    17738                     10 files
```

Let's run it in an Android Emulator (Figures 1 and 2):

```
$ ./adb install /tmp/insomnidroid/insomnidroid.apk
240 KB/s (15076 bytes in 0.061s)
        pkg: /data/local/tmp/insomnidroid.apk
Success
```

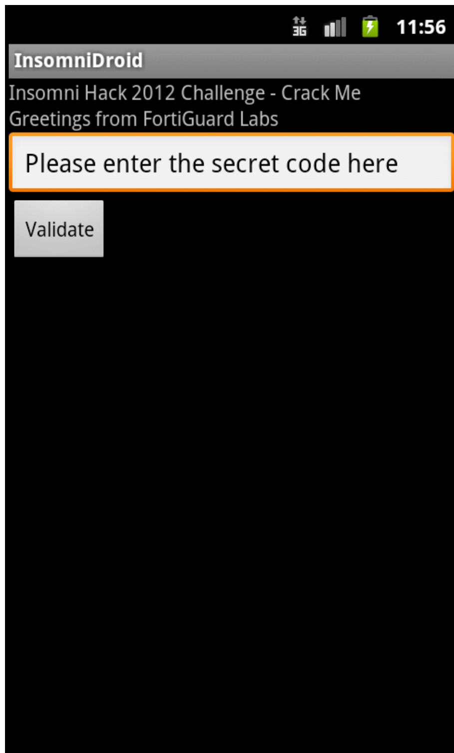

Figure 1: CrackMe's main screen

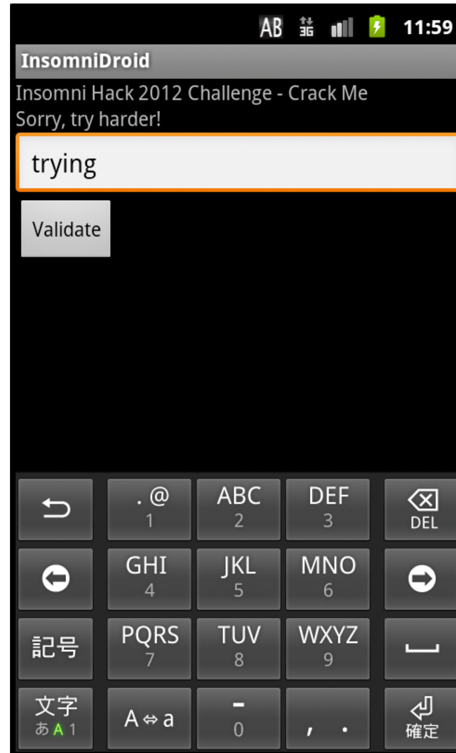

Figure 2: Wrong password: "try harder"

So, it's all about finding the right password.

3 Reverse engineering

3.1 Decompilation

There are plenty of ways to reverse Android applications [Apv12], let's just "convert" it to a jar with dex2jar [Dex]:

```
$ ~/softs/dex2jar-0.0.9.6/dex2jar.sh insomnidroid.apk
dex2jar version: reader-1.7, translator-0.0.9.6, ir-1.4
dex2jar insomnidroid.apk -> insomnidroid_dex2jar.jar
Done.
```

Now, we open the jar with a Java decompiler such as [Jdg], and try to understand the code (Figure 3). The code has obviously been obfuscated (several dummy names).

3.2 Android Manifest

There are three classes: InsomniActivity, a and b (see left side of Figure 3). What are they for, which one is the main entry point? To find out, we first unzip the APK and convert its AndroidManifest.xml to a human-readable format:

```
$ java -jar ~/softs/AXMLPrinter2.jar AndroidManifest.xml
```

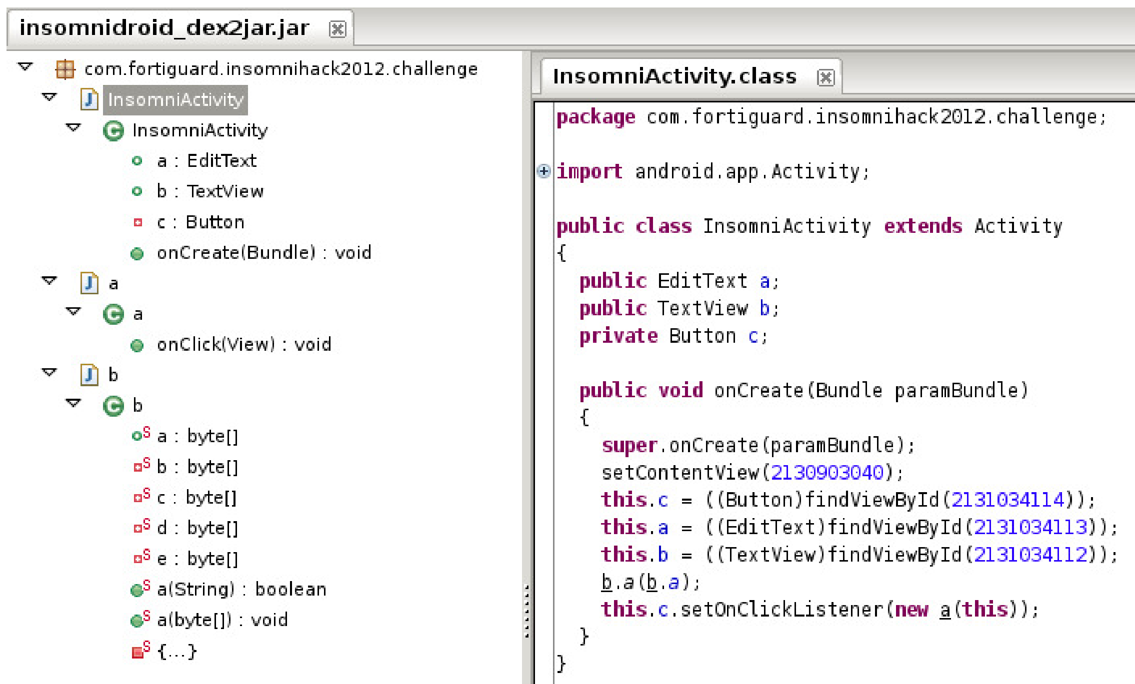

Figure 3: Reversed InsomniActivity class

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:versionCode="1"
        android:versionName="1.0"
    package="com.fortiguard.insomnihack2012.challenge">
  <application android:label="@7F040001"
               android:icon="@7F020000">
    <activity android:label="@7F040001"
              android:name=".InsomniActivity">
      <intent-filter>
        <action        android:name="android.intent.action.MAIN">
        </action>
        <category android:name="android.intent.category.LAUNCHER">
        </category>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Lines 1-15 of the manifest show InsomniActivity is the entry-point. There are no receivers or services.

3.3 Reversing InsomniActivity

We try to understand the short code of InsomniActivity (see Figure 3). It is quite short:

- it displays the frame (text, edit, button)

- it calls a *method* b.a(), providing *field* b.a as parameter
- it instantiates a click listener named "a", which is activated when the button "Validate" is pressed

3.4 Reversing a - and a lazy solution

```
InsomniActivity.class | b.class | a.class
package com.fortiguard.insomnihack2012.challenge;

import android.text.Editable;

final class a
  implements View.OnClickListener
{
  a(InsomniActivity paramInsomniActivity)
  {
  }

  public final void onClick(View paramView)
  {
    if (!b.a(this.a.a.getText().toString()))
    {
      this.a.b.setText("Sorry, try harder!");
    }
    else
    {
      this.a.b.setText("Congrats! -- FortiGuard Team");
      this.a.a.setEnabled(false);
    }
  }
}
```

Figure 4: Decompiled class a

This class is very simple. When the button "Validate" is pressed, the method onClick() is called. It retrieves the string the end-user entered (this.a.a.getText().toString()) and calls method b.a() for that string. If the result is false, the application displays text "Sorry, try harder!", otherwise it displays the congratulations message.

At this point, an option would be to hack the APK, modify its smali code [Sma] and recompile it. For instance, we could negate the test that checks the string, i.e change the `if-eqz` bytecode to `if-nez` (which is the opposite). See [Apv12] for details on how to do that. Then, whatever string we enter (except the good one!) displays the congratulations message.

However, as this is a challenge CrackMe, modifying the application isn't fun. Rather, we'll try and find the password.

3.5 Reversing b

The only other class we haven't inspected yet is class b. This class contains:

- 5 byte arrays
- a method a() that takes a String as input parameter and responds with a boolean. This is the validation method (see 3.4).

- another method a() that takes a byte array as input and has no output. This method is called by InsomniActivity (see 3.3).

Let's have a closer look at the validation method (see code below). The method initiates the SHA-256 hash function, digests the input and checks it against static field b. If both digests are equal, the method returns true (good message). The other peculiarities of the method are due to decompilation and should not be taken into account.

```
public static boolean a(String paramString) {
    try   {
      Object localObject = MessageDigest.getInstance("SHA-256");
      ((MessageDigest)localObject).reset();
      localObject = ((MessageDigest)localObject).digest(paramString.getBytes());
      boolean bool = Arrays.equals(b, localObject);
      if (bool)      {
        bool = true;
        return bool;
      }
    }
    catch (Exception i)    {
      while (true)    {
        Log.w("InsomniDroid", "checkSecret: " + localException.toString());
        int i = 0;
      }
    }
  }
```

Above in class b, we find the value of constant b:

```
arrayOfByte = new byte[32];
arrayOfByte[0] = 97;
arrayOfByte[1] = 82;
...
b = arrayOfByte;
```

So, we need to find a text whose sha256 matches that one. Cracking or brute-forcing SHA-256? Let's hope we don't have to, this hash function hasn't any known vulnerability. See [Aum12] for more information on hash functions and current issues.

Let's now have a look at the other method a():

```
1  public static void a(byte[] paramArrayOfByte)  {
2      try    {
3        SecretKeySpec localSecretKeySpec = new SecretKeySpec(paramArrayOfByte, "AES");
4        IvParameterSpec localIvParameterSpec = new IvParameterSpec(e);
5        Cipher localCipher = Cipher.getInstance("AES/CTR/NoPadding");
6        localCipher.init(2, localSecretKeySpec, localIvParameterSpec);
7        localCipher.doFinal(c);
8        byte[] arrayOfByte = new byte[d.length];
9        for (int i = 0; i < d.length; i += 16) {
10         localCipher.init(2, localSecretKeySpec, localIvParameterSpec);
11         localCipher.doFinal(d, i, 16, arrayOfByte, i);
12       }
13     }
14     catch (Exception localException)  {
15       Log.w("InsomniDroid", "compute: " + localException.toString());
16     }
17   }
```

This method performs some AES computations. More precisely,

1. it uses AES CTR, where CTR is a special mode of operation that uses a counter [Dwo01]
2. the input parameter is the key
3. it sets the counter with the value of static field e
4. it initializes the cipher object (localCipher) and deciphers static value c (line : 2 = DECRYPT_MODE)
5. it allocates an array of 80 bytes (d.length = 80)
6. for each 16-byte block, it initializes the cipher object and deciphers d into the newly allocated array.

On a crypto point of view, there is something awfully wrong in the description I just made, but for the sake of the spoiler, let's say we haven't spotted it yet ;)

This method is called by InsomniActivity, with *field a* as parameter.

```
byte[] arrayOfByte = new byte[16];
arrayOfByte[0] = -34;
arrayOfByte[1] = -83;
arrayOfByte[2] = -66;
..
a = arrayOfByte;
```

Let's try and print this array in characters, as much as possible (for example, using the python interpreter in interactive mode with the following commands – credits Alexandre Aumoine):

```
>>> from array import *
>>> array('b', [-34, -83, -66, -17, 101, 112, 105,
  99, 32, 102, 97, 105, 108, 1,2,3]).tostring()
>>> a=[-34, -83, -66, -17, 101, 112, 105,
  99, 32, 102, 97, 105, 108, 1,2,3]
```

The output is:

```
'\xde\xad\xbe\xefepic fail\x01\x02\x03'
```

It reads out "(0x)deadbeef" and later "epic fail". This does not look like a valid key, but rather dummy data with possibly some hints.

So, to summarize, at this point, we have:

- a SHA256 hashing function to validate the password and check it against a constant pre-computed hash.
- an AES-CTR decryption function, without the real key. This method is quite strange, because it has no output. It operates on buffers c and d, and does not seem to be used elsewhere by the application.

3.6 Crypto

Let's get back to the description of method a(). It is strange that the cipher object is *initialized for each block*. We would have expected:

1. initialize the cipher object
2. for each block, decipher the block

Do we care? well, yes, because apart from being slow and stupid, it also means that the initialization data (localIvParameterSpec - see line 4) is going to be the same for each block. And for AES CTR, initialization data is the counter, so the application is using the same counter for each block, and, as a matter of fact, also for c!

Cryptographers know that the CTR mode of operation is only secure if counters are never re-used! [Dwo01]:

> *"The sequence of counters must have the property that each block in the sequence is different from every other block. This condition is not restricted to a single message: across all of the messages that are encrypted under the given key, all of the counters must be distinct."*

On [Wik], we can read:

"*For OFB and CTR, reusing an IV completely destroys security.*"

Wow! It seems really bad! Why? Let's have a look to definition of CTR (don't worry there's no complicated maths). We consider we have some plaintext P which consists in several block P_1, P_2 .. P_n. $\oplus$ is the XOR operator. Each block of ciphertext C is computed as follows:

$$C_i = AES_{key}(counter_i) \oplus P_i \tag{1}$$

So, if all counters$_i$ are the same, we have:

$$\forall i, counter_i = counter \tag{2}$$

$$C_i = AES_{key}(counter) \oplus P_i \tag{3}$$

In practice, this means that:
$C_1 = AES_{key}(counter) \oplus P_1$
$C_2 = AES_{key}(counter) \oplus P_2$
..
$C_n = AES_{key}(counter) \oplus P_n$
Note that $AES_{key}(counter)$ is *constant*. Consequently, the differences between two blocks of ciphertexts is equal to the difference of plaintexts:

$$\begin{aligned} C_1 \oplus C_2 &= (AES_{key}(counter) \oplus P_1) \oplus (AES_{key}(counter) \oplus P_2) && (4)\\ &= (AES_{key}(counter) \oplus AES_{key}(counter)) \oplus (P_1 \oplus P_2) && (5)\\ &= P_1 \oplus P_2 && (6) \end{aligned}$$

If one of the plaintext P_i is null, then a simple XOR with its block of ciphertext provides the plaintext (no need to know the key nor the counter).

$$C_i = AES_{key}(counter) \oplus null, \Longrightarrow \forall j, C_i \oplus C_j = P_j \tag{7}$$

In the challenge, we have two ciphertexts: c (a single 16-byte block) and d (5 block of 16 bytes). Let's try and XOR c with each block of d and see what happens. In the best case, we'll get some plaintext (if c or a block of d corresponds to the enciphering of null). In the worst cases, we'll get information on the differences between each block of plaintext. That'll be useful for cryptanalysis anyway.

The Java code below does the XOR, with input values copy pasted from the fields of class b. Use any other method or language of your convenience to perform the XOR.

```
public class ChallengeSoluce {
    public static byte [] c = new byte[16];
    public static byte [] d = new byte[80];

    static {
     c[0] = -20;
     c[1] = 52;
     c[2] = 39;
     ..
     c[15] = 33;
     d[0] = -81;
     d[1] = 91;
     ..
     d[79] = 23;
    }
    public static byte [] doXor(byte [] a, byte [] b) {
        byte [] c = new byte[b.length];
        int i, j;
        for (i=0,j=0; j<b.length; i++,j++) {
            if (i >= a.length) i = 0;
            c[j] = (byte) (a[i] ^ b[j]);
        }
        return c;
    }

    public static void main(String args[]) {
        byte [] value = ChallengeSoluce.doXor(c,d);
        System.out.println("Result: "+new String(value));
    }
}
```

We compile and then run the computation:

```
$ java ChallengeSoluce
Result: Congrats! Dont re-use AES CTR counters ;) Secret Code is: 2mkfmh2r0hkake_m123456
```

We're lucky! We get the solution straight away! This means that ciphertext c corresponded to the enciphering of a null plaintext.

We enter **2mkfmh2r0hkake_m123456** in the CrackMe (see Figure 5).

By the way, "epic fail" was a reference to the PS3's hack [bmss10]: researchers managed to retrieve the playstation's private key due to bad implementation of crypto - like in this challenge.

4 Other Write-Ups

- Tim Strazzere [Str12] tested the challenge for me, and posted a nice write-up (after the challenge ;)). He actually found the solution a bit by luck, because he hadn't spot the implementation issue of CTR.
- SCRT [Ins12] published several solutions to other challenges

5 Acknowledgements

Thanks to Alexandre Aumoine, David Maciejak, Tim Strazzere for trying the CrackMe, and SCRT for organizing Insomni'Hack.

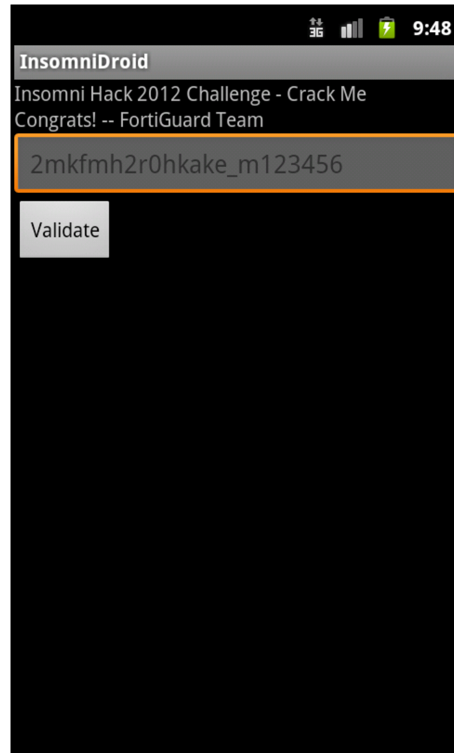

Figure 5: CrackMe solved!

References

[Apv12] Axelle Apvrille. Android Reverse Engineering Tools, March 2012. Insomni'hack, http://www.fortiguard.com/sites/default/files/insomnidroid.pdf.

[Aum12] Jean-Philippe Aumasson. Insomni'Hash, March 2012. Insomni'hack, http://www.131002.net/data/talks/insomnihack12_slides.pdf.

[bmss10] bushing, marcan, segher, and sven. Console hacking 2010, ps3 epic fail. In *27th Chaos Communication Congress*, December 2010. http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf.

[Dex] Dex2jar. http://code.google.com/p/dex2jar/.

[Dwo01] Morris Dworkin. *Special Publication 800-38A: Recommendation for block cipher modes of operation*. National Institute of Standards, U.S. Department of Commerce, December 2001.

[Ins12] Insomni'Hack 12, March 2012. http://www.scrt.ch/insomnihack/2012/presentation.

[Jdg] Java Decompiler. http://java.decompiler.free.fr/.

[Sma] Smali. https://code.google.com/p/smali.

[Str12] Tim Strazzere. InsomniDroid crackme solution, March 2012. http://www.strazzere.com/blog/?p=488.

[Wik] Block Cipher Modes of Operation. http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation.