Infection Risks of a Smartwatch

Eurecom Semester Project

Abstract

There is more and more concern about security. Connected devices widen the attack surface. For our semester project, we aimed to analyze the security aspects of the Sony Smartwatch 2 and more precisely the risks of infection by malwares that already exist in the Android ecosystem. Thus, in this report, we deal with how Smartwatch apps work, Android app patching and repackaging, Bluetooth dumping and other amazing security and Android related topics.

Students:

Axel Ehrenstrom
Msc student at Eurecom
Communication and System Security

Soufiane Joumar
Msc student at Eurecom
Communication and System Security

Table of contents

1. Introduction

This few last years, we are seeing the emergence of what are called connected objects. Connected by WI-FI or Bluetooth, they have access to sensitive data (like personal messages, credentials etc.) and generate new sensitive data (such as geolocation, biometric measurements). Time and budget pressure have involve a lack of security in the data communication and storage.

Connected objected are by definition linked to another device, mainly to mobiles. Mobiles are subject to malwares and according to Kaspersky, they have found more than 100 000 new malwares targeting mobiles only in 2013. For the most part, targets are Android OS based mobiles. The underlying question we want to answer here is: can Android malware infect a Sony Smartwatch 2 (subject of the study) ? How ?

First, we take an interest in the functioning of our product - Sony SmartWatch 2, its apps Smartwatch and SmartConnect and how they communicate. Then we study the attack surface, and see where are the security lacks and how we can exploit them to extract data and get a deeper understanding o the product's functioning. We finally suggest some further possibilities for future development and research.

2. SmartWatch functioning

By Axel

2.1. The Watch capabilities

The watch only access to the network is using Bluetooth. Therefore, the watch can only retrieve information that are provided by the Bluetooth device to which it is connected. For example, the watch cannot download the facebook feed associated with an account. The Bluetooth device have to download it itself and then send it via bluetooth to the watch. The watch will then be able to display it.

Also the watch only have a ARM Cortex-M3 CPU and will not allow the user to do any computation on it. The SmartWatch extensions are installed on the buetooth device and all the computation is done on it. The watch will only display the output of this computation. Thus if you appaired your SmartWatch to another bluetooth device you will loose all your previous applications and gain the one installed on the new bluetooth device as the application are bound to them.

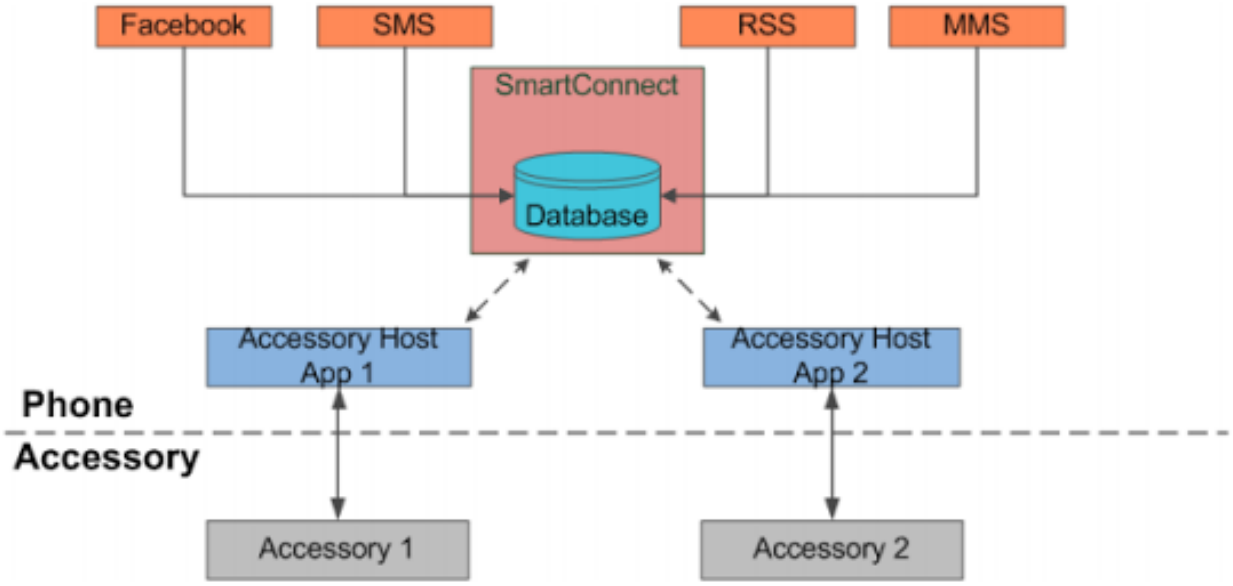
Provided that we concluded that the watch could only show content, through touch events (touch-based finger activity) or key events (key press activity).

Therefore, the watch has to be connected the a Bluetooth device to be “smart”. When bought the watch is discoverable on the bluetooth network and can be appaired with any bluetooth device using SmartConnect. The watch then become hidden and cannot be appaired to a second bluetooth device. To change the bluetooth device to which it is appaired, the watch has to be reset using the Settings option. It then forget the appaired device and is become discoverable on the bluetooth network allowing another appairing.

2.2. Interactions between the watch and the bluetooth device

The SmartWatch provides information about the tactical screen (touch events and key events) and other sensors (light and acceleration). Based on these information, the application can print content. The applications that use these features needs to be connected to the bluetooth device in order to function and are disabled otherwise.

Another way to show content is to use the notification API. The functioning of this API can be resumed in this graphic :



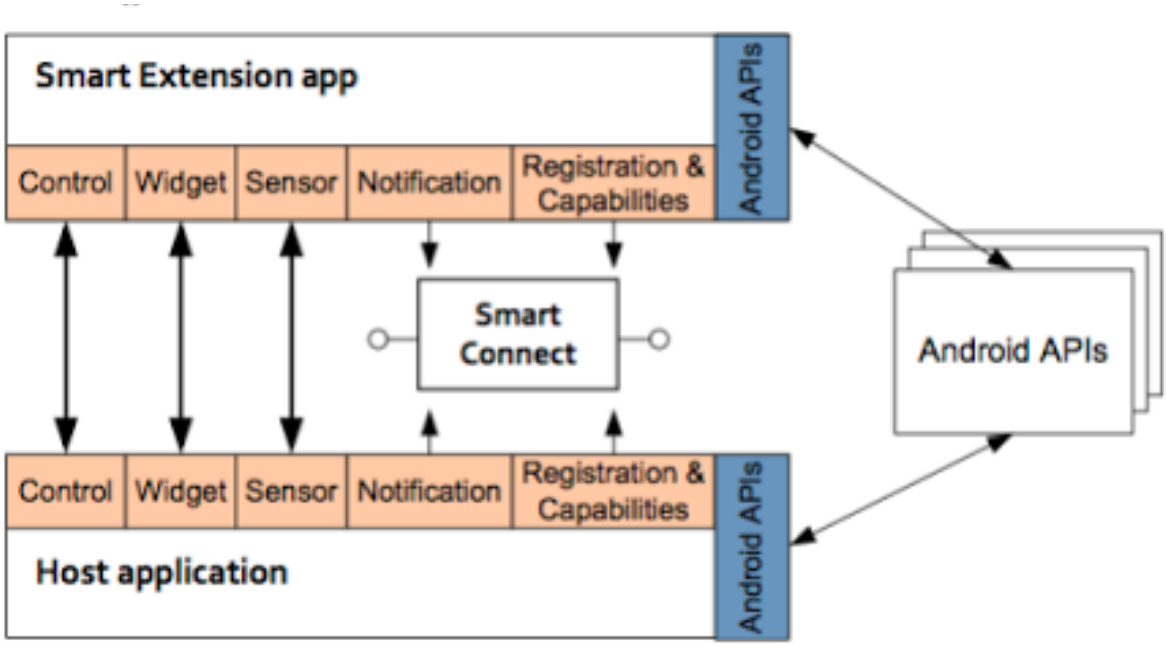
The SmartExtension get some content : an SMS, a facebook message or whatever and want to show it on watch. The process requires two steps :

- the SmartExtension creates an event which is add in a content provider in SmartConnect.
- The SmartWatch application (Accessory Host)retrieve the event and display it on the watch.

The main advantage of this method is that is does not required the smartphone to be always connected. During the connection the events are kept up to date but when offline the watch can still display the events that have been previously loaded.

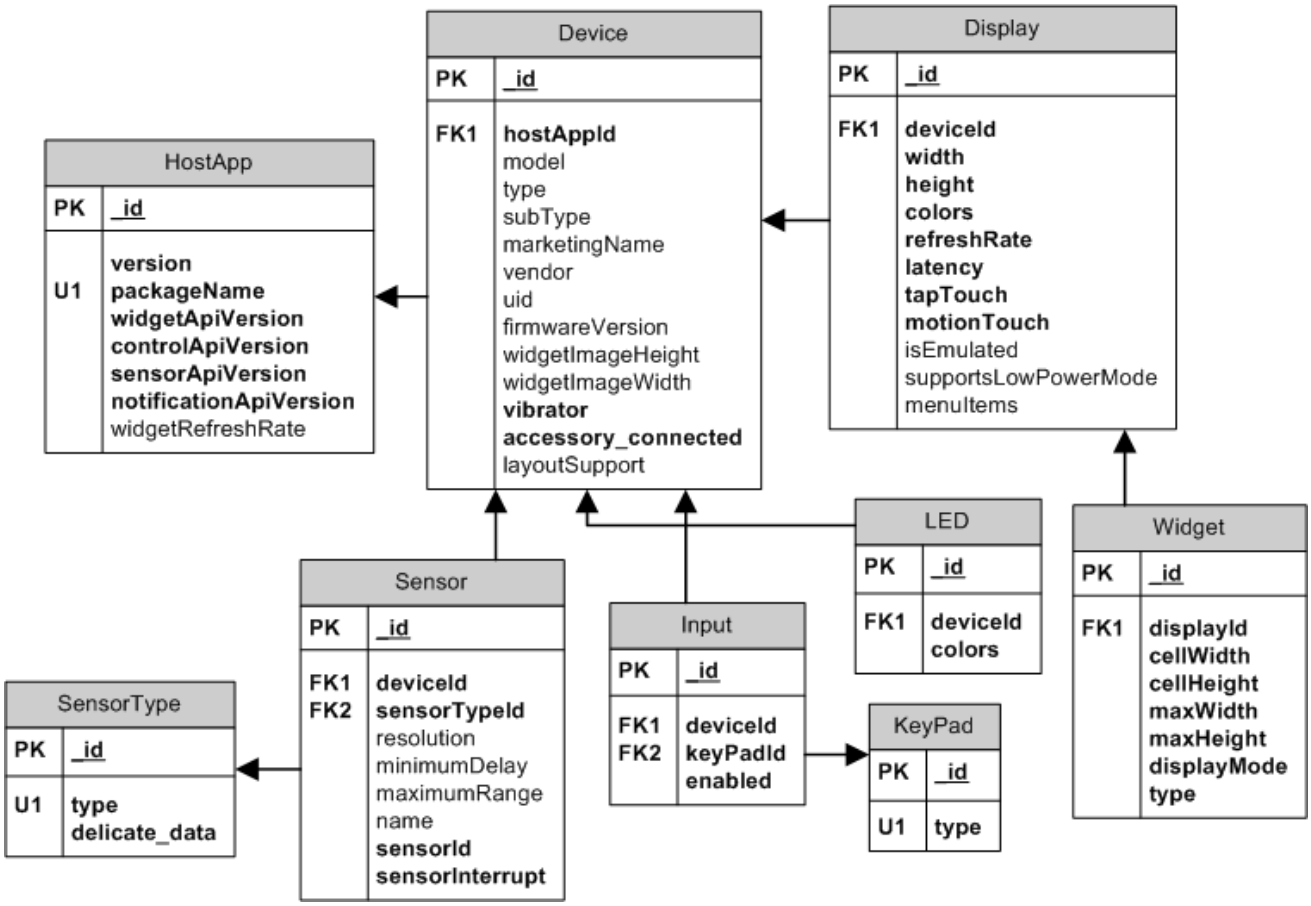
2.3. The SmartExtensions fonctioning

The interaction between a SmartExtension and SmartConnect/SmartWatch 2 can be resumed in the following graphic :

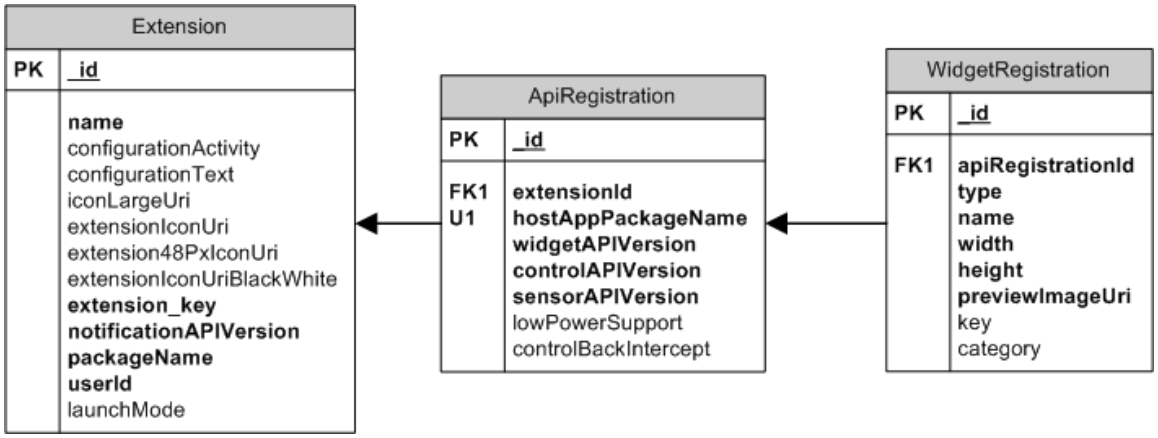


Capabilites and Regsitration API works this way :

When a host application like SmartWatch is installed on the smartphone, it insert informations about the capabilities of the smartwatch in a content provider. This informations will be used by the smartphone to interact with the watch :



When SmartWatch is installed, and therefore has provided its capabilities, every time a new application is installed the intent EXTENSION_REGISTER_REQUEST_INTENT is broadcast. If the application is a smartExtension it has to answer to that intent and insert a record in the extension tables :



This record must contain the API it will use, a package name and so on. After that steps the SmartEnxtensions can use the following API :

- Control, Widget and sensor API : the basic features of the watch that the SmartExtensions can acess through smartwatch 2
- Notification API : the extension can start writing sources and events in the notification database.

3. Attack surface

During our discovery phase, we were interested in several points of the Smartwatch where we studied the possibility of infection/vulnerabilities. Thus, we chose to take an interest in exploring the Bluetooth communication to determine the security level of the Bluetooth stack, studying how the messages (notifications) between the smartwatch host app and the watch are packed by the host app. We also looked for ways to implements SmartExtensions that had a malicious behavior. Besides, we performed a firmware analysis, the goal being learning which C library (for the bluetooth protocol or for displaying images on the watch's screen for example) are used and how to use them in order to make the watch do action that it is not supposed to, in order to disable the

Bluetooth security protection and let another device be connected to the watch and leak data for example .

3.1. SmartExtensions

By Axel

Here we took already existing SmartExtensions and tried to modify them, so that they had a malicious behavior. As the watch is only a remote screen, in the way that the programmer cannot execute any code inside nor send directly some data, we had to perform the malicious activity on the phone. The idea being to use the watch to trigger some malicious behaviour on the phone.

We used a sample smartExtension provided by Sony created to show the use of the sensor : at launch the application show the accelerometer data, on click on the screen, the application swap the le light sensor data. We used the onClick() method to perform additional malicious actions : every time the user swap the application send a message to a paying number.

This is the only type of attack scenario that we could imagine using SmartExtensions.

3.2. Bluetooth communication

By Soufiane

In this part, we began by determining the level of security of the Bluetooth implementation. Sony Smartwatch 2 uses Bluetooth 3.0 which is a quite old version of Bluetooth (2009) especially for connected objects.

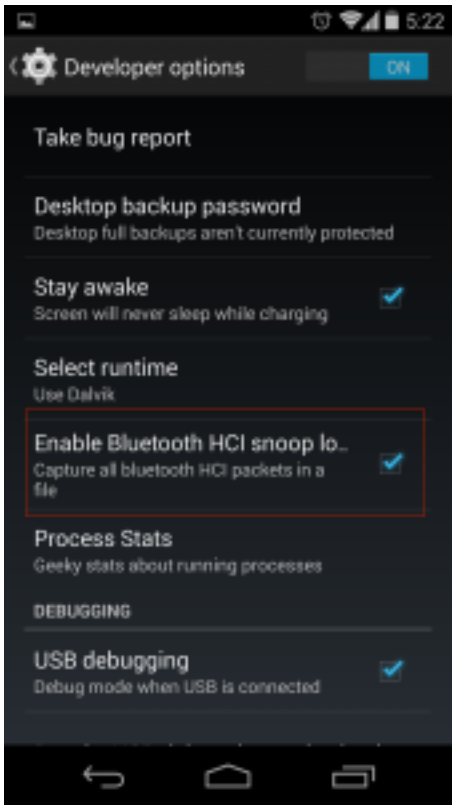
In Bluetooth, there are three security mode:
Mode 1: no specific security.
Mode 2: service level security, above data link layer.
Mode 3: data link level security.

So we tried to determined the security level of the Bluetooth protocol implemented by Sony.

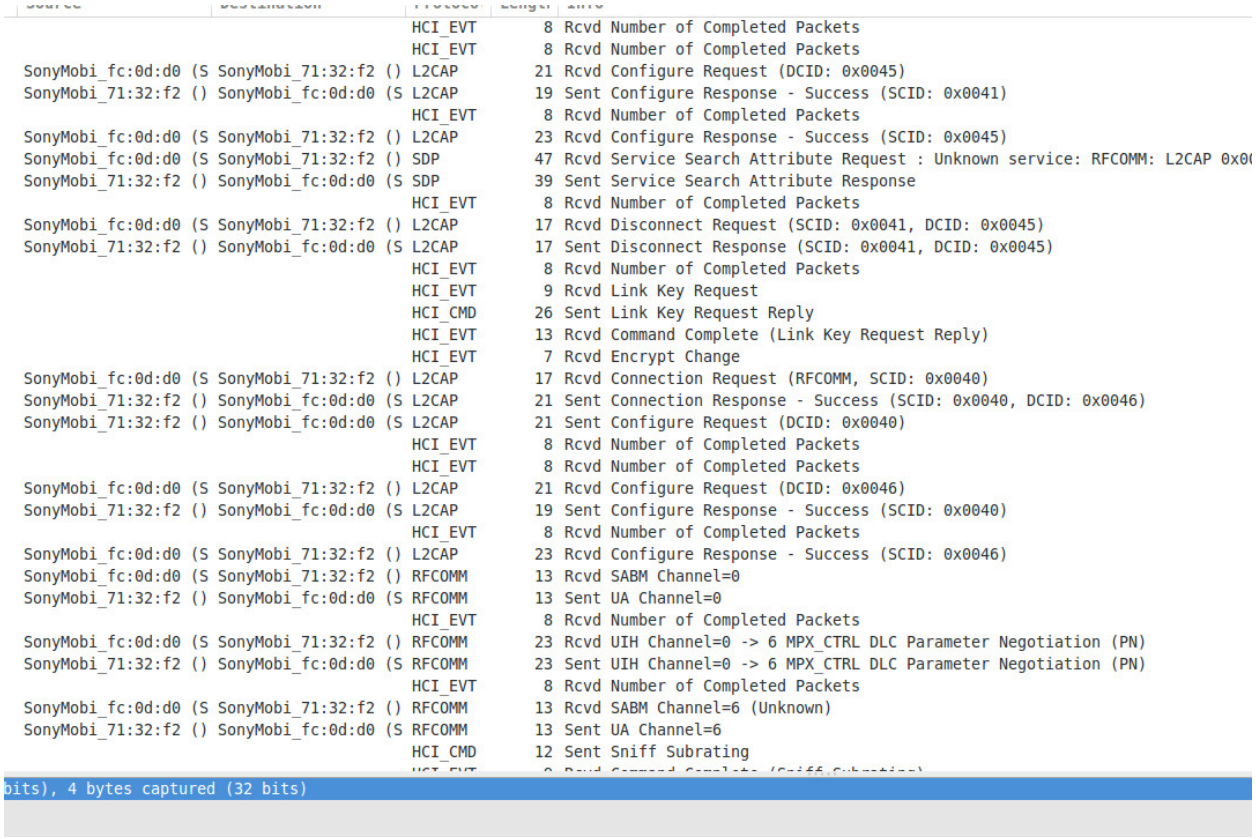
3.2.1. Bluetooth Sniffing

First of all, we had to sniff the Bluetooth packets communicating between the phone and the Sony watch. Ho w this can be done ? Either with a Bluetooth dongle sniffer or you can also do it with Android 4.4+ which has a native feature that can actually sniff HCI (Host Controler Interface) Bluetooth packets going from and to the smartphone and send them to a file.

You can access this feature by enabling the Android development mode and activate the Bluetooth capturing. When finished, disable the capturing and you can get the dump file in the external storage (e.g SD card).



This generated file can be analyzed with a packet analyzer like Wireshark. This is what you can see then:



		HCI_EVT	8	Rcvd Number of Completed Packets
		HCI_EVT	8	Rcvd Number of Completed Packets
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	L2CAP	21	Rcvd Configure Request (DCID: 0x0045)
SonyMobi_71:32:f2 ()	SonyMobi_fc:0d:d0 (S	L2CAP	19	Sent Configure Response - Success (SCID: 0x0041)
		HCI_EVT	8	Rcvd Number of Completed Packets
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	L2CAP	23	Rcvd Configure Response - Success (SCID: 0x0045)
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	SDP	47	Rcvd Service Search Attribute Request : Unknown service: RFCOMM: L2CAP 0x0(
SonyMobi_71:32:f2 ()	SonyMobi_fc:0d:d0 (S	SDP	39	Sent Service Search Attribute Response
		HCI_EVT	8	Rcvd Number of Completed Packets
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	L2CAP	17	Rcvd Disconnect Request (SCID: 0x0041, DCID: 0x0045)
SonyMobi_71:32:f2 ()	SonyMobi_fc:0d:d0 (S	L2CAP	17	Sent Disconnect Response (SCID: 0x0041, DCID: 0x0045)
		HCI_EVT	8	Rcvd Number of Completed Packets
		HCI_EVT	9	Rcvd Link Key Request
		HCI_CMD	26	Sent Link Key Request Reply
		HCI_EVT	13	Rcvd Command Complete (Link Key Request Reply)
		HCI_EVT	7	Rcvd Encrypt Change
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	L2CAP	17	Rcvd Connection Request (RFCOMM, SCID: 0x0040)
SonyMobi_71:32:f2 ()	SonyMobi_fc:0d:d0 (S	L2CAP	21	Sent Connection Response - Success (SCID: 0x0040, DCID: 0x0046)
SonyMobi_71:32:f2 ()	SonyMobi_fc:0d:d0 (S	L2CAP	21	Sent Configure Request (DCID: 0x0040)
		HCI_EVT	8	Rcvd Number of Completed Packets
		HCI_EVT	8	Rcvd Number of Completed Packets
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	L2CAP	21	Rcvd Configure Request (DCID: 0x0046)
SonyMobi_71:32:f2 ()	SonyMobi_fc:0d:d0 (S	L2CAP	19	Sent Configure Response - Success (SCID: 0x0040)
		HCI_EVT	8	Rcvd Number of Completed Packets
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	L2CAP	23	Rcvd Configure Response - Success (SCID: 0x0046)
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	RFCOMM	13	Rcvd SABM Channel=0
SonyMobi_71:32:f2 ()	SonyMobi_fc:0d:d0 (S	RFCOMM	13	Sent UA Channel=0
		HCI_EVT	8	Rcvd Number of Completed Packets
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	RFCOMM	23	Rcvd UIH Channel=0 -> 6 MPX_CTRL DLC Parameter Negotiation (PN)
SonyMobi_71:32:f2 ()	SonyMobi_fc:0d:d0 (S	RFCOMM	23	Sent UIH Channel=0 -> 6 MPX_CTRL DLC Parameter Negotiation (PN)
		HCI_EVT	8	Rcvd Number of Completed Packets
SonyMobi_fc:0d:d0 (S	SonyMobi_71:32:f2 ()	RFCOMM	13	Rcvd SABM Channel=6 (Unknown)
SonyMobi_71:32:f2 ()	SonyMobi_fc:0d:d0 (S	RFCOMM	13	Sent UA Channel=6
		HCI_CMD	12	Sent Sniff Subrating

bits), 4 bytes captured (32 bits)

We do not think that a link level security is established, it is the bluetooth mode 2 - service level security. There is still authentication procedure and encryption. Encryption is based on a shared key generated from a link key that has been exchanged between the phone and the watch.

This shared key (varying length key (8-128 bits), regenerated for each transmission from link key) is stored in the security manager (part of the Bluetooth chip). The shared key needs also a PIN number, but we never entered any PIN number during connection procedures between the SW2 and the phone.

So we guess there is probably a fixed PIN number stored (up to 128 bit). A path could be to bruteforce the PIN or to have an access to it in order to pair another device to this watch and get data from it.

Conclusion:

Data are transferred from the phone to the watch and the data is encrypted: sms, mails, social networks notifications. We need at least the link key if we want to obtain the data. In order to leak data from a phone-watch communication we would need some Bluetooth penetration techniques. But it falls out of this project's scope. See 4.3 for more some options concerning Bluetooth exploits.

We just have seen what could be done concerning Bluetooth as an attack vector. As a consequence, we wanted to understand how the packets are built at the application level, meaning by the host app Smartwatch. How did we investigate into that ?

We used reverse engineering techniques to obtain the java source code of the host app Smartwatch.

3.2.2. Smartwatch Android app reverse engineering

Some tools were used to achieve this part:

- **apktool**: a tool for android reverse engineering of binary Android files.
- **smali**: intermediate language between .dex bytecode and actual java code. It is used to assemble/disassemble android softwares.
- **jarsigner**: android apps are signed. Thanks to a key and a certificate, jarsigner generates a digital signature.
- **adb**: a tool to perform app installs, uninstalls, debugging, android shell etc.

A android app written in java is compiled in JVM bytecode and then this bytecode is transformed at runtime into Dalvik (VM) bytecode to be executable (Dalvik Executable: .dex or odex) in any hardware setting that has Android as OS.

That being said, you can easily reverse engineer an Android app and get the java source code. There will certainly be some errors but you can work on it to understand how the app is coded.

Thus, to patch an Android app, you cannot write in java and re-compile it because of those errors. You will have to patch by writing smali code and recompile it with apktool. To use the app, you need to resign the app with jarsigner before recompiling it.

General steps to patch an apk file:

- decompile the apk with apktool
- edit the smali files obtained in the previous file
- sign the directory using jarsigner
- compile it with apktool
- install it with adb

3.2.3. Host app patching

3.2.3.1. Why patching

We chose to patch the Sony host app ‘Smartwatch’ in order to understand how it works, and to dump two things :

- notifications’ messages
- Bluetooth payloads before processed by the Bluetooth chip

3.2.3.2. Where to patch

When you decompile the Smartwatch host app and obtain the source code, you get these directories:

```
code/sw2_hostapp/com/sonymobile/smartconnect/hostapp$ ls
analytics    costanza     fota                  preferences  util
config       Dbg.java     HostAppAefConfig.java protocol
Config.java  debugevents  layout                sensor
connection   extensions   notification          service
```

So if we look at the file ./notification/EventManager.java, we can see that there is a method called readSmartConnectEvent() that returns an 'Event' object. This is where we are going to patch in order to get notifications' messages like sms, mails etc.

Let's now have a look at the file ./connection/CommunicationManager.java, you can see that a 'write' method is implemented. This is the method that sends Costanza messages to the sending buffer queue that leads to the bluetooth chip. Here we can dump packed Costanza messages into the logcat console.

3.2.3.3. How to patch an Android app

In this part, we are going to explain in more details how you can inject smali code into decompiled apk Dalvik bytecode.

First, as we previously said, we have to decompile the apk using apktool and its option 'd'. Let's say it is called *smartwatch.apk* . The command line would be here:

```
apktool d smartwatch.apk -o output_directory/
```

Then you can access the smali files in the output_directory and edit them.

The smali syntax is quite readable and you can easily do simple operations. But since you want to perform more complex functions, you might need to first write them in java compile/decompile them to obtain the smali corresponding code.

I will describe here the smali patching for getting the notifications' messages and writing them in an externally stored file.

Here is the beginning of readSmartConnectEvent() method in the smali file:

```
.method private readSmartConnectEvent(Landroid/database/Cursor;)Lcom/sonymobile/smartconnect/hostapp/notification/Event;
    .locals 22
    .param p1, "c"    # Landroid/database/Cursor;
    .annotation system Ldalvik/annotation/Throws;
        value = {
            Ljava/io/IOException;
        }
    .end annotation
    .prologue
    .line 471
    const-wide/16 v2, -0x1

    .line 472
    .local v2, "id":J
    const-string v1, "_id"

    move-object/from16 v0, p1

    invoke-interface {v0, v1}, Landroid/database/Cursor;->getColumnIndexOrThrow(Ljava/lang/String;)I

    move-result v1

    move-object/from16 v0, p1

    invoke-interface {v0, v1}, Landroid/database/Cursor;->getLong(I)J

    move-result-wide v8

    .line 473
    .local v8, "smartConnectEventId":J
    const-string v1, "sourceId"

    move-object/from16 v0, p1

    invoke-interface {v0, v1}, Landroid/database/Cursor;->getColumnIndexOrThrow(Ljava/lang/String;)I

    move-result v1
```

You can notice that all java objects are stored in 32 register (e.g. v0) , p1 being the method parameter (here a 'Cursor' object) . Here 22 registers are reserved for the method. Registers can be overridden.

We coded in java the snippet code that gets the string we want and write it in an output file in the phone specified by a path. We compiled it by using apktool and its option 'b':

```
apktool b script.java
```

We got the corresponding apk using eclipse and we decompile it again in order to obtain the following smali code:

```
new-instance v1, Ljava/io/File;
const-string v14, "/storage/emulated/0/Download/dump.txt"
invoke-direct {v1, v14}, Ljava/io/File;-><init>(Ljava/lang/String;)V
.local v1, "file":Ljava/io/File;
new-instance v15, Ljava/io/FileWriter;
const/4 v0, 0x1
invoke-virtual {v1}, Ljava/io/File;->getAbsoluteFile()Ljava/io/File;
move-result-object v14
invoke-direct {v15, v14, v0}, Ljava/io/FileWriter;-><init>(Ljava/io/File;Z)V
.local v15, "fw":Ljava/io/FileWriter;
new-instance v0, Ljava/io/BufferedWriter;
invoke-direct {v0, v15}, Ljava/io/BufferedWriter;-><init>(Ljava/io/Writer;)V
.local v0, "bw":Ljava/io/BufferedWriter;
invoke-virtual {v0, v13}, Ljava/io/BufferedWriter;->write(Ljava/lang/String;)V
invoke-virtual {v0}, Ljava/io/BufferedWriter;->close()V
```

The message we want to dump is stored in the object referenced by the string ‘Message’ stored in v13. We just need to add this snippet code after the result have been stored in v13. Meaning after:

```
invoke-interface {v0, v1}, Landroid/database/Cursor;->getString(I)Ljava/lang/String;
move-result-object v13
```

Now, the smali code being patched, we can recompile it and test it. But first we have to resign it. Since SmartConnect has a signature based verification scheme we have to re-sign also SmartConnect so that both SmartConnect and Smartwatch are signed with the same key and certificate. Android does not verify the origin nor the integrity of the first installation of the app. Indeed, Android verifies only if an update matches the original app by checking the signatures.

To resign the the package SW2, supposing I am at Resources/SonySmartWatch2/hostapp_modification, I run :

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1
-keystore ../debug.keystore sw-mod.apk androiddebugkey
```

Then we can recompile the app to get the apk file:

```
apktool b smali_directory/ -o patched_smartwath.apk
```

You can now download the apk on the phone and install it using adb.

3.2.3.4. Results

We can see that each time there is an event (launching an extension on the Smartwatch), the hos app read all the messages stored on the SmartConnect's database related to the launched app and we could also print other fields like the message id, status or the corresponding contact. Here we mainly see text messages I received. It prints up to the last 100 messages.

As said above, we performed a similar patch in a lower level method in the Bluetooth stack. We managed to print the packed Costanza (name of the 'protocol' implemented by Sony) messages' Bluetooth payloads. I patched the file connection/CommunicationManager.java, the method write().

As a result, we get plain ASCII characters in which we can see the messages we just mentioned above:

```
)125070  c0 80 0d 0a 56 2f 6c 6f  67 63 61 74 20 20 28 20  |....V/logcat  ( |
)125080  39 35 36 33 29 3a 20 ef  bf bd c0 80 c0 80 c0 80  |9563): .........|
)125090  01 c0 80 c0 80 c0 80 2f  05 c0 80 c0 80 ef bf bd  |......./........|
)1250a0  04 10 c0 80 c0 80 c0 80  c0 80 c0 80 c0 80 c0 80  |................|
)1250b0  c0 80 c0 80 6e c0 80 c0  80 c0 80 c0 80 c0 80 c0  |....n...........|
)1250c0  80 c0 80 c0 80 c0 80 c0  80 c0 80 c0 80 c0 80 c0  |................|
)1250d0  80 c0 80 09 c0 80 c0 80  c0 80 c0 80 c0 80 c0 80  |................|
)1250e0  c0 80 ef bf bd c0 80 12  c0 80 01 c0 80 c0 80 c0  |................|
)1250f0  80 03 c0 80 c0 80 c0 80  01 c0 80 c0 80 c0 80 04  |................|
)125100  c0 80 c0 80 c0 80 04 c0  80 c0 80 c0 80 c0 80 c0  |................|
)125110  80 c0 80 c0 80 c0 80 c0  80 1d c0 80 c0 80 c0 80  |................|
)125120  4e c0 80 6f c0 80 75 c0  80 76 c0 80 65 c0 80 6c  |N..o..u..v..e..l|
)125130  c0 80 6c c0 80 65 c0 80  73 c0 80 20 c0 80 76 c0  |..l..e..s.. ..v.|
)125140  80 69 c0 80 64 c0 80 ef  bf bd c0 80 6f c0 80 73  |.i..d.......o..s|
)125150  c0 80 20 c0 80 64 c0 80  69 c0 80 73 c0 80 70 c0  |.. ..d..i..s..p.|
)125160  80 6f c0 80 6e c0 80 69  c0 80 62 c0 80 6c c0 80  |.o..n..i..b..l..|
)125170  65 c0 80 73 c0 80 20 c0  80 c0 80 c0 80 0d 0a 56  |e..s.. ........V|
)125180  2f 6c 6f 67 63 61 74 20  20 28 20 39 35 36 33 29  |/logcat  ( 9563)|
)125190  3a 20 ef bf bd c0 80 c0  80 c0 80 01 c0 80 c0 80  |: ..............|
)1251a0  c0 80 2f 05 c0 80 c0 80  ef bf bd 04 10 c0 80 c0  |../.............|
)1251b0  80 c0 80 c0 80 c0 80 c0  80 c0 80 c0 80 c0 80 6e  |...............n|
)1251c0  c0 80 c0 80 c0 80 c0 80  c0 80 c0 80 c0 80 c0 80  |................|
```

We can see on these screenshot that packets are not encrypted by Sony and between each character there are 2 bytes c0 80. Indeed you can read : “Nouvelles videos disponibles” which is a notification from the Youtube Smartwatch extension.

Let’s try to determine how those Costanza messages are built and packed before being sent to the Bluetooth chip.

3.2.4. Costanza packets and protocol

The Android Smart Watch Host app uses a native system library to deal with Bluetooth communication. This library is simply called ‘protocol’. Indeed you can see that here, at /Resources/SonySmartWatch2/smartwatch_app_sourcecode/sw2_hostapp/com/sonymobile/smartconnect/hostapp/protocol/Proto.java

```
ublic abstract class Proto

static
{
  System.loadLibrary("protocol");
}

public Proto()
{
  init();
}

private native void init();

protected synchronized native byte[][] pack(CostanzaMessage paramCostanzaMessage);
```

You can see that a native method is needed to pack and unpack Costanza message, i.e. transforming Bluetooth data into Smartwatch object called Costanza message.

A Costanza message is simply built with a buffered pack (byte [][]), the message’s type and message id.
See here :

```
public abstract class CostanzaMessage
{
  private byte[][] bufferedPack;
  private int messageId;
  protected int type;
```

3.3. Firmware

Now that we have a complete understanding of how the Bluetooth is managed and how data is packed by Smartwatch host app, let's focus a bit on the firmware. To obtain it, we had to dump it from the hardware device.

3.3.1. Firmware dumping

By Soufiane

To obtain the firmware, we stumbled upon this tutorial:
http://lunarius.fe80.eu/blog/smartwatch2-first-steps.html

Here are the main steps:

- Shutdown your smartwatch
- Disconnect usb cable
- only connect micro usb connector to smartwatch
- you have around 1sec for this: press the powerbutton, plug the other usb cable end to the computer and release the powerbutton

Then you will have to upload the firmware from the watch to the computer with a tool called dfu-util.
Thus, we get a 2Mo file from the Flash internal memory that we can now analyze.

3.3.2. Analysis

By Axel

To analyze the firmware we first tried basics commands to try to find some immediate information :

- strings

- hexedit

We then tried to open it with ida pro but it didn't recognized any function.

While looking for information about the firmware we found an official release of the firmware by Sony and tried to confront the result of the previous methods on the new firmware but the result were roughly the same.

After running an entropy test. We realized that the firmware was partly compressed. And that's why we couldn't find many instructions.

The analysis of binwalk first revealed nothing but with the its version 2.0, binwalk revealed some promising zlib.

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
56055         0xDAF7          Zlib compressed data, compressed
655620        0xA0104         LZMA compressed data, properties: 0xC0, dictionary size:
706639        0xAC84F         Zlib compressed data, compressed
764961        0xBAC21         Zlib compressed data, default compression
783902        0xBF61E         Zlib compressed data, default compression
827024        0xC9E90         Zlib compressed data, compressed
854972        0xD0BBC         Zlib compressed data, best compression
```

We extracted these Zlib independently and also considering that they were just one Zlib and tried to decompress them. We used both shell command and a python library but in both case it failed. The header of Zlib is only two bytes and lead to these false positives.

We also looked for obfuscation : we realized that the 4 bytes of the .zip header were present at multiple occasion with just a little inversion. We we restored them and tried to uncompress but it was also a mislead.

While looking for some leads about the firmware we found a forum were somebody had extracted the bootloader from the firmware. The bootloader could be open with IDA PRO.

But in front of the complexity of the bootloader we decided that we couldn't go any further.

3.3.3. Conclusion

We managed to dump the firmware and tried to analyze it. We didn't managed to get any information out of it but we showed that is was possible to understand it by finding the bootloader. We decided to drop it here because of the time constraints.

4. Opening and future possibilities

By Soufiane

4.1. Firmware modifications

Since we did not have the time for exploring more the firmware files, we were not able to override them. It can be useful to enable some C functions for malicious reasons (retrieve data for example) , to allow a malicious code or allow a phone to connect to the watch that is not supposed to connect to.

4.2. Hardware hacking

The hardware setting is pretty basic and simple, we can imagine leak data from the watch by analyzing the memory or the CPU .

4.3. Bluetooth exploits

Since it was not in the scope, we did not perform Bluetooth penetration hacks. But it can be interesting to achieve a Bluetooth vulnerability

assessment to know what can be done in the phone's physical area with attacks like Bluesnarfing, Bluebugging, Bluejacking or even DOS attacks.

5. Conclusion

We have been through a number of experiments and research to understand and test our Sony Smartwatch product. We came to the conclusion in the first part that due to its lack of performance and power, its small and limited firmware, traditional Android malwares will not target this product. Indeed, we can consider this Smartwatch as a second little screen for the phone, being able to print to last messages, mails etc. through notifications.

However, it was the occasion for us to widen the project's scope, by assessing the Bluetooth attacks options, understanding the Sony's Costanza protocol and patch its Android application (by using reverse engineering interesting techniques), exploiting extensions capabilities to make the user do non unintentional actions like sending a message to a another phone. Thus, Smartwatch app can be developed as an SMS trojan repackaged app. We were also able to perform a firmware analysis that would require a bit more time to be profitable, in order to hack C libraries that were found in the firmware. We let this work for future possibilities.

6. References

Android Malware

- Kaspersky statistics analysis of malwares in 2013: http://securelist.com/analysis/kaspersky-security-bulletin/58265/kaspersky-security-bulletin-2013-overall-statistics-for-2013/#01
-

Android app repackaging and reverse engineering

- Smali development documentation: http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html
- How to use apktool: http://forum.xda-developers.com/android/general/guide-apk-tool-ubuntu-complete-t2834788
- Android reversing: http://blog.indiandragon.in/2012/10/ultimate-android-reverse-engineeringdecompiling-guide-part-1-softwares-and-procedure.html
- Sign an Android app: http://developer.android.com/tools/publishing/app-signing.html#releasecompile
- Obtaining source code and Android app patching: http://blogmal.42.org/rev-eng/patching-android-apps.story

Bluetooth - Bluetooth Security

- Bluetooth architecture presentation by Bluetooth Special Interest Group: http://srohit.tripod.com/bluetooth_sec.pdf
- NSA recommandations about Bluetooth secure development: https://www.nsa.gov/ia/_files/factsheets/i732-016r-07.pdf
- NIST Guide to Bluetooth Security: http://csrc.nist.gov/publications/nistpubs/800-121-rev1/sp800-121_rev1.pdf

Firmware Dumping and Analysis

- Firmware dumping :

http://wiki.openmoko.org/wiki/Manuals/dfu-util
http://lunarius.fe80.eu/blog/smartwatch2-first-steps.html
http://forum.xda-developers.com/showthread.php?t=2783426&page=4

- Firmware analysis :
http://binwalk.org/
https://media.blackhat.com/us-13/US-13-Zaddach-Workshop-on-Embedded-Devices-Security-and-Firmware-Reverse-Engineering-WP.pdf